

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF





**Broadview**<sup>®</sup>  
www.broadview.com.cn



虫术

# Python绝技

梁睿坤◎著

/ 代码兼顾Python 2和Python 3 / 分享实战项目源代码 /  
/ 深入分析爬虫测试与调试过程 / 详解可视化爬虫 /



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>





梁睿坤 ▶

近二十年软件开发、项目管理、团队建设和管理经验。致力于互联网技术应用与大数据应用方面的研究与开发工作。曾任多家软件公司的高级软件工程师、项目经理、首席架构师和技术总监等职务。

现任广州市增增智能科技有限公司CEO，从事视觉智能、语音智能及IoT等技术的产品研发与企业经营方面的工作。







容内職全職代價士社本其性我陳壁方式臨云心研不 何有則未 育正器



# Python绝技

梁睿坤◎著

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING





## 内 容 简 介

本书以大数据应用方面常用的语言 Python 为基础,从网络爬虫的实现原理入手,逐步引领读者进入网络爬虫的世界。在各类爬虫框架中,将 Scrapy 作为轴心,从多个维度揭开爬虫技术的面纱。例如,爬取规则的制定技巧,设计高速爬虫,如何让爬虫更“聪明”地获取数据,将海量数据进行分布式存储的技术,设计具有高隐匿性的爬虫,大规模、高并发的分布式爬虫技术。

本书基于 Python 这门灵活且简洁的语言,结合作者在网络数据爬取和大数据方面的实际工程经验,使得本书更具实用性。本书旨在让更多数据工作者或编程爱好者在大数据时代从海量的信息中通过掌握“虫术”来获取对自己或企业有价值的信息。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有,侵权必究。

## 图书在版编目(CIP)数据

虫术: Python 绝技 / 梁睿坤著. —北京: 电子工业出版社, 2018.7  
ISBN 978-7-121-34456-5

I. ①虫… II. ①梁… III. ①软件工具—程序设计 IV. ①TP311.561

中国版本图书馆 CIP 数据核字(2018)第 125215 号

责任编辑: 陈晓猛

印 刷: 三河市双峰印刷装订有限公司

装 订: 三河市双峰印刷装订有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16 印张: 26.75

字数: 513.6 千字

版 次: 2018 年 7 月第 1 版

印 次: 2018 年 7 月第 1 次印刷

定 价: 99.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式: 010-51260888-819, [faq@phei.com.cn](mailto:faq@phei.com.cn)。







# 前言

很久以前我就接触了网络爬虫这门技术，从当时接触的范畴来说，称之为“小玩意”或者“小助手”可能更为贴切。我使用爬虫只是为了收集一些样本数据做测试，或者对上线的项目进行高强度的并发性压力测试，又或者获取感兴趣的图片、新闻。

爬虫涉及的技术比较多，用各种语言都可以快速地写出一个爬虫，所以一直以来并没有被看作一门综合性的技术，直到 2015 年我负责的开发部门接到公司安排的三项重点开发任务：

(1) 从微信和微博上搜集哪些言论正变得热门，哪些公众号或者微博账号的关注度正在持续地上升。

(2) 要与一家技术很落后的电商公司的业务系统在没有提供数据接口的情况下进行大规模的数据同步。

(3) 开发一个数据可视化平台，并导入公司内部多年来的销售数据（都是一些 Excel 和 CSV 文件），然后将当前每月在京东、淘宝等电商平台上的统计数据合并起来进行统一的查询与统计。

在接到这三个任务时，可以说是没有任何头绪的，这些任务简单看都是一些数据整合的工作。在深入分析与研究之后发现，要完成这三大任务都必须依赖爬虫技术。

这是一个坑坑洼洼，而且充满挑战的过程。例如，如何能从号称封闭独立的微信中挖出数据，又不被屏蔽；如何能将每天过亿条的数据存储下来而不会“塞爆”服务器；如何能将每天一大堆的 CSV 或者 Excel 文件下载到服务器，然后自动整理入库而不会出现数据错误，等等。在完成这三个项目之后，我和我的团队都对爬虫有了非常深刻的理解与认识，很多方面的知识 & 经验都得到了极大的提高。在综合过往的开发经验和这几年的实际入坑经验之后，我决定将其编撰成书，将这些看似零散的技术融合起来。





## 内容介绍

“虫术”是一门综合性的技术，涉及的知识面很广，为了不让你在一大堆的技术面前感到茫然，我将这门“术”分成了三个运用阶段，一步步由浅入深地进行叙述。

本书共 5 章，前 3 章为初阶部分，第 4 章为中阶部分，第 5 章为高阶部分。

### 第 1 章 爬虫初步

本章首先介绍爬虫在目前大数据生态下的地位，还提供了一份关于学习虫术的详尽的技术线路图，最后讲述爬虫基本的实现方法与实际运用示例，目的在于让读者对虫术建立一个基本的概念并能从示例中引起对这门技术的兴趣。

### 第 2 章 Scrapy 基础

虫术以 Scrapy 架构为核心基础，本章对 Scrapy 的架构和各个模块的作用进行了详细的介绍。

### 第 3 章 Scrapy 工程管理与部署

本章介绍如何在 Scrapy 工程中运用 Scrapyd 将本地工程部署到实际运行环境中，详细地讲述 Scrapyd 安装配置及其附带的 scrapy-client 和 scrapy-deploy 工具的使用方法。

### 第 4 章 中阶虫术

本章包含的内容非常丰富，是针对将虫术运用于实际项目展开的。从 Scrapy 的蜘蛛内部实现开始，深入 HTTP 底层，实现对 Scrapy 中间件的支持，运用 Selenium 或 Splash 处理棘手的 JavaScript 网页，最后详细讲述如何处理采集到的数据。

### 第 5 章 高阶虫术

本章是对中阶虫术的深化，聚焦于爬虫系统的性能，讲解如何让爬虫变得更加隐蔽，如何让爬虫看懂图片，如何训练它们使之变得更加聪明，最后讲解如何掌握虫术的大招“分布式爬虫”来应对大规模的数据集采工作与数据存储任务。

## 勘误

本书如有勘误，会在 <https://github.com/DotNetAge/>上发布。由于笔者能力有限，时间仓促，书中难免有错漏，欢迎读者批评指正。

梁睿坤







## 读者服务

轻松注册成为博文视点社区用户（[www.broadview.com.cn](http://www.broadview.com.cn)），扫码直达本书页面。

- **下载资源：**本书如提供示例代码及资源文件，均可在[下载资源处](#)下载。
- **提交勘误：**您对书中内容的修改意见可在[提交勘误处](#)提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方[读者评论处](#)留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34456>





# 入门必读——浅谈 Python

这是一本完全基于 Python 技术体系的图书。如果你没有学过 Python，那么你需要先找几本 Python 入门的基础书籍来看看。Python 语言本身就不是以一书之篇幅能将其内容所尽述的，毕竟术业有专攻，如果你是 Python 的初学者，则可以从以下几本书中选一本先行学习：

- *Head first Python*, O'REALY Paul Barry
- *Programming Python*, O'REALY Mark Lutz
- *Learning Python*, O'REALY Mark Lutz

## Python 世界

即使你学习过 Python，但如果你还没得到 Python “血统”与“传承”，那么本书对你来说可能还是非常苦涩难懂的。幸运的是，Python 的“血统”与“传承”不是继承的，而是通过后天得到的，也是用来衡量是否完全进入 Python 世界的门槛。Python 是一门很有意思的语言，它来源于自由开放的开源世界，因此它没有被任何一家富有而强大的企业所霸占，也不像那些来自“软甲谷”<sup>1</sup>一系身披光华。与其他主流语言相比，Python 显得非常纯粹、简美又具有智慧。“开放”是 Python 血统的一大特色，正因为开放，所以它成为现今世界上使用最为广泛的语言之一！使用 Python 的人不只有 IT 界，在 Python 的官方社区 [pypi.org](https://pypi.org) 上的大量极为出名的第三方软件包并不是出于我们软件人之手，它们来源于数学、化学、天文、物理等各种科学计算领域。一个真正的 Python 程序员面对一个问题第一思考方向从来不是如何去写一段代码来解决它，而是在社区里找一下有没有第三方包可以解决这样的问题，我将此称为“开放的血统”。

---

<sup>1</sup> 软甲谷：此处暗喻微软、甲骨文与谷歌。







## 关于哲学

什么是传承？传承不但是—种技艺或者技术，还应该是一种思想方法和哲学思维。你可能会疑惑：一个软件工程师也需要哲学思维？当然，一个普通的软件工程师是不需要什么哲学思维的，正如普通的音乐演奏者并不需要什么音乐天赋—样。但如果你爱软件开发，希望在此路上走得更远，设计哲学、开发哲学会是指引你前行的明灯。

谈到哲学，估计你会觉得像读大学上思想政治课或者哲学理论课那样无趣、难懂。恰恰相反，Python 的作者 Tim Peters 早就在 Python 中种下了他的思想种子，只要你在 Python 的命令行内输入 `import this`，就能看到以下内容：

```
>>> import this
The Zen of Python, by Tim Peters—Python 的设计哲学，作者：Tim Peters
```

Beautiful is better than ugly.——优雅胜于丑陋。

Explicit is better than implicit.——明确胜于含糊。

Simple is better than complex.——简单胜于复杂。

Complex is better than complicated.——复杂胜于烦琐。

Flat is better than nested.——扁平胜于嵌套。

Sparse is better than dense.——间隔胜于紧凑。

Readability counts.——可读性很重要。

Special cases aren't special enough to break the rules.——即使假借特殊之名，也不应打破这些原则。

Although practicality beats purity.——尽管实践大于理论。

Errors should never pass silently.——错误不可置之不理。

Unless explicitly silenced.——除非另有明确要求。

In the face of ambiguity, refuse the temptation to guess.——面对模棱两可，拒绝猜测。

There should be one--and preferably only one --obvious way to do it.——用—种方法，最好只有—种方法来做—件事。

Although that way may not be obvious at first unless you're Dutch.——虽然这种方式开始时并不明显，除非你是 Python 之父。

Now is better than never.——从现在开始这么做总比永远都不做好。

Although never is often better than \*right\* now.——尽管经常“没有做”反倒比“现在马上做”的结果要好。

If the implementation is hard to explain, it's a bad idea.——如果—个实现方案不容易解释，那么它肯定不好。



If the implementation is easy to explain, it may be a good idea.——如果一个实现方案很容易解释, 那么它也许是个好主意。

Namespaces are one honking great idea -- let's do more of those!——就像命名空间就是一个绝妙的想法, 应当多加利用。

哲学是理论与实践的边界, 设计哲学就是方法论。理解正确的设计哲学才会让我们在使用 Python 时更深切地领略到其优秀与精华之所在。

## 学习 Python 的方法

学习 Python 不会像学习其他语言那么复杂, 因为它很直接、很简洁。快速读完它的语法手册, 你就已经可以轻松地在 Python 的世界中畅游了。Python 的发展得益于其开放而庞大的社区。它是一门用于解决问题的语言, 在某个问题域内你总可以找到一套优秀的开源工具包甚至是一套出色的开发框架。

因此, 我认为学习 Python 的最佳路径是:

- (1) 发现问题, 分析其关键。
- (2) 找对解决问题的工具包或者框架。
- (3) 以解决问题为目的, 一边实践, 一边学习。

对技术始终保持一颗好奇的心, 有空多上 GitHub 下载感兴趣且 Stars 和 Forks 都比较多的 Python 项目仔细地阅读, 从别人的代码中学习。积极参与感兴趣的开源项目, 与全球的贡献者一起工作与交流, 这样会让你的 Python 技艺得到长足的进步与发展。



# 目 录

第 1 章 爬虫初步.....	1
1.1 爬虫与大数据 .....	1
1.1.1 大数据架构 .....	1
1.1.2 爬虫的作用与地位 .....	3
1.1.3 Python 与爬虫 .....	5
1.1.4 Python 的网络爬虫框架 .....	6
1.1.5 虫术技术路线图 .....	9
1.2 实例：简单的爬虫 .....	10
1.3 内容分析进阶 .....	13
1.3.1 选择器 .....	15
1.3.2 深入 BeautifulSoup .....	16
1.3.3 元素的搜寻 .....	18
1.3.4 乱码与中文编码 .....	22
1.4 新闻供稿的爬取实例 .....	31
1.5 小结 .....	39
第 2 章 Scrapy 基础知识.....	40
2.1 Scrapy 架构 .....	41
2.2 Scrapy 快速入手 .....	43
2.3 数据模型 Item .....	47

2.4 蜘蛛——Spiders .....	51
2.5 管道——Item Pipeline .....	54
2.6 Scrapy 的运行与配置 .....	60
2.7 新闻供稿爬虫的 Scrapy 实现 .....	63
2.8 小结 .....	65
 第 3 章 Scrapy 的工程管理 .....	66
3.1 Scrapy .....	67
3.2 scrapyd-client 及部署 .....	77
3.3 搭建爬虫服务器 .....	81
 第 4 章 中阶虫术 .....	87
4.1 蜘蛛的演化 .....	88
4.1.1 蜘蛛的本质——深入 Spider .....	88
4.1.2 通用蜘蛛 .....	93
4.1.3 蜘蛛中间件 .....	117
4.2 爬虫系统的测试与调试 .....	128
4.2.1 开发期调试 .....	128
4.2.2 蜘蛛的测试 .....	131
4.2.3 蜘蛛的运行期调试 .....	133
4.2.4 调试内存溢出 .....	139
4.3 处理 HTTP 请求 .....	144
4.3.1 HTTP 请求 .....	145
4.3.2 Scrapy 的 Request 对象 .....	156
4.3.3 表单处理 .....	162
4.3.4 下载器中间件 .....	164
4.4 处理 HTTP 响应 .....	177
4.4.1 HTTP 响应 .....	178
4.4.2 Scrapy 的响应对象 .....	182
4.4.3 深入选择器 .....	184



4.4.4 非结构化数据的提取 .....	196
4.4.5 黑夜中的眼睛 .....	217
4.5 处理 JavaScript .....	229
4.5.1 示例：电商产品爬虫 .....	230
4.5.2 Selenium 和 PhantomJS .....	235
4.5.3 Scrapy 与 Splash .....	259
4.6 数据存储与后处理 .....	292
4.6.1 图片的下载与存储 .....	293
4.6.2 示例：产品图片采集 .....	298
4.6.3 导出到数据文件 .....	299
4.6.4 导出到数据库 .....	302
4.6.5 示例：基于阿里云的存储后端 .....	308
 第 5 章 高阶虫术 .....	 324
5.1 增量式爬网 .....	325
5.1.1 推演路由 .....	326
5.1.2 时机的重要性 .....	328
5.1.3 去重处理 .....	329
5.1.4 布隆过滤器 .....	331
5.1.5 基于 Redis 的布隆过滤器 .....	336
5.2 突破封印 .....	340
5.2.1 封禁浅析 .....	343
5.2.2 客户端仿真 .....	346
5.2.3 化身万千——蜘蛛世界的易容术 .....	356
5.2.4 反跟踪 .....	377
5.2.5 绕开蜜罐 .....	380
5.3 虫海 .....	385
5.3.1 分布式爬虫架构 .....	385
5.3.2 认识 scrapy-redis .....	386
5.3.3 示例：分布式电商爬虫 .....	390



5.4	可视化爬虫 .....	393
5.4.1	示例: 某点评网爬虫 .....	397
5.4.2	解读 Portia 爬虫代码 .....	402
5.4.3	数据项加载器——Item Loaders .....	410
5.4.4	最后的工作 .....	413

# 1 chapter

## 第 1 章 爬虫初步

本章介绍爬虫的相关概念与基本运用，包括爬虫在大数据时代所占据的地位和发挥的作用，还有用于开发爬虫的语言 Python，以及在 Python 下的一些爬虫框架。

另外，爬虫涉及的技术领域很多，运用的技术也非常庞杂，从基本的网络访问到复杂的机器学习，可能会让初入门径者有望而却步的想法。为了能让读者对虫术保持一颗好奇的心，不至于被知识的海洋淹没，本章的最后一节提供了一份学习虫术的详细技术路线图，按照实际的学习与运用的阶段划分了需要涉猎与掌握的各种相关技术。

本章最后从实践出发，通过简单的爬虫浅尝一下虫术，希望通过实践能让读者对这门技术有一个基本的概念与认识。

### 1.1 爬虫与大数据

网络爬虫是一种伴随着互联网诞生与演化的“古老”的网络技术，我记得第一次听到“爬虫”这个词还是在早年的雅虎网站上（谷歌还没出生）。从那时起，搜索引擎就是运用网络爬虫在各类网络资源中爬取关键字或者一些简单的内容描述来建立网站索引目录的，爬虫技术的发展可以说是与互联网发展同步的。随着互联网进入大数据时代，爬虫技术迎来了一波新的振兴浪潮。

#### 1.1.1 大数据架构

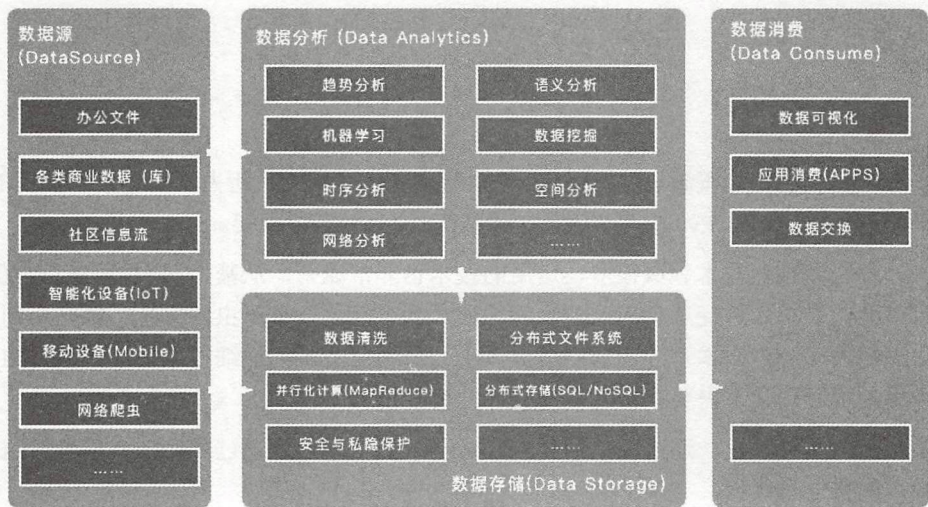
爬虫在大数据架构中有着举足轻重的位置，在对其进行详细讲述之前先来聊聊大数据。在



我的印象中，大数据（Big Data）的概念大约是从 2007 年前后开始兴起的，当时并不觉得这是很“高大上”的技术，毕竟我以前做的项目没有哪个是没有数据库的。由于我对各种技术都抱有强大的好奇心，所以从那时起，除了本职工作需要使用 SQLServer 和 Oracle，其他数据库也都喜欢研究一番，以至于对大数据的浅层次理解就是：

应用程序的运行数据多了自然就成为“大数据”了。

显然，这种理解是非常粗浅的，虽说数据量大是“大数据”应用的一种表面的特征，但这并不是全面的。因为“大数据”是一种采用多种技术对数据进行综合性处理的系统化简称。如果将大数据应用采用层次理论简单地分层展开，则会让我们有一个比较客观和感性的认识。下图是按照功能的作用与数据的供给对大数据应用进行分层。



大数据应用分为以下几个层次。

- 原始数据源（Data Source）——负责进行数据的采集、过滤去重、清洗、修复，按照实际应用需要对数据进行规范化处理，可以直接将数据提供给分析加工层进行深加工或将数据提供给持久层进行存储。
- 数据分析与加工层（Data Analytics）——负责对原始数据进行深加工，例如，文本语言分析、情感分析、传播路径分析、机器学习或人工智能化分析，并将数据分析结果提供给消费层使用。
- 存储层（Data Storage）——对结构化、非结构化数据进行存储。针对大型分散式应用还可以提供分布式的数据存储。
- 消费层（Data Consum）——负责对外部提供数据展现或者自动化数据供给，也就是多



维度的数据可视化和各种数据 API。

- 基础架构层（Infrastructure）——为大数据应用提供系统级的支持，例如，建立数据集群、虚拟化服务等。

这个层次图是我对大数据应用的一个笼统的概括，由于每个层次中的一个小部分都足以用一本书甚至是一个系列书的篇幅进行叙述，已经远远超出了本书的讨论范围。我画出此图的最重要的原因是希望读者能关注其中的“原始数据源”与“存储层”两个层次，它们除了是大数据架构中的重要组成部分，也是本书所讲述的重点——爬虫。

### 1.1.2 爬虫的作用与地位

在大数据架构中，数据收集与数据存储占据了极为重要的地位，可以说是大数据的核心基础。而爬虫技术在这两大核心技术层次中占有了很大的比例。为何有此一说？我们不妨通过一个实际应用场景来看看爬虫到底发挥了哪些作用？

在了解爬虫的作用之前，应该先了解其基本特性：

- **主动**——爬虫的重点在于“爬取”（Crawl），这是一种主动性的行为。换句话说，它是一个可以独立运行且能按照一定规则运作的应用程序。
- **自动化**——由于处理的数据可能很分散，数据的存留具有一定的时效性，所以它是一套无人值守的自动化程序。

#### 企业内部爬虫

在我接近 20 年的 IT 从业生涯中，企业管理系统是我参与过的项目或产品中占比最大的。在这些项目与产品的开发过程中，我观察到很多企业内部其实有非常多的数据处理场景可以用爬虫技术进行处理，从而能以惊人的效率取代原有的人工化的操作。

以我近年来在电商企业内部所见为例，阿里巴巴（简称阿里）已显现出它在电子商务一统全球的实力与地位，几乎可以将电商与阿里之间划一个等号。阿里为各个店铺和商家提供了各种各样优秀的运营工具。我们会理所当然地认为电商企业内部的信息化管理程度一定很高，不是吗？然而事实恰恰相反，我见过的多数中小型的电商企业甚至是三板挂牌企业内部的信息化水平仍然非常落后，不少企业仍然依赖 Excel 这样基于大量人力为主导的表格处理。那么问题来了，为何阿里巴巴、京东这些电商平台已经提供了大量优质运营工具，而电商企业的信息化水平却很低，还需要靠劳动密集型的方式进行运营呢？首先，电商企业不会只在某一平台上开店，通常都会在多个平台同时开多个店铺以拓宽市场的销售渠道；其次，电商企业之间、电商与供货商之间缺乏统一的数据交换标准，通常只依赖于一些技术陈旧的 ERP 来维持日常的运营。

电商企业通常只能通过某一平台上提供的专用工具监测某些产品的价格波动和销售情况，

而无法全面、统一地了解他们所销售的产品在各大平台的具体表现如何。然而这样的需求很明显是迫切的,因为只有了解销售数据的变化才能实时调节销售的策略。我见过最多的做法就是企业安排一位专人从各大电商平台中导出运行的数据,然后合并到 Excel 中,再进行一番统计,手工做出各种统计报表作为分析依据,这种做法往往对某一个单品就得做一次!

其实,导致这种现象的原因有以下几点:

- (1) 缺乏统一的数据来源——这是不可调和的,因为电商运行的数据源本身就具有多样性。
- (2) 结构化数据与非结构化数据并存——企业间最常见的数据交互格式是 Excel,交互工具是微信和 QQ。
- (3) 一个数据存在多种时间版本——QQ 或者微信上的同一个文件修改多了且重复传会出现各种的 `data.xlsx`、`data(1).xlsx`...`data(n).xlsx`。
- (4) 数据结构可能存在随意性——Excel 文件内很少会看见用英文命名的列,甚至相同作用的列很有可能会采用不同的中文名。
- (5) 数据查找变得困难——在电商企业与供货商之间要找出某个时段相同的数据副本可能是一件极为可怕的事件。

我们不妨来大胆地假设一下,如果将这些事情换成让爬虫去处理,那么情况会变成什么样子呢?

- (1) 每天爬虫在一个固定的时间到淘宝、京东或者其他电商平台上自动下载商家当前的营业数据。
- (2) 完成爬取后将数据自动保存到数据库。
- (3) 从内网的某台 PC 的指定文件夹中下载每天从其他经销商发来的 Excel 文件,整理后保存到数据库。
- (4) 发现某些商品库存不足自动生成供货商规定格式的订货单,通过电子邮件发出。
- (5) 决策者(运营经理/老板)在手机或 PC 中通过数据可视化工具查看每天的数据统计结果,或者由爬虫系统直接生成统计报表发到他们的邮箱中。

此时你可能会产生这样的疑问:爬虫不是单单爬取数据的吗?为何还能处理这么多的事情呢?这还是爬虫的技术领域吗?答案是肯定的,上面这个例子是由我经历过的一个项目中的真实案例简化而来的,爬虫的这些行为融合了对爬取数据的后处理与 Python 自动化后得到的效果。其实爬虫能做到的事情可以更多,具体的实现与企业内部的实际需求相关。而在互联网中,它更像是一个具有“智能”的机器人。关于这些内容在本书后面的章节会有具体且详细的讲述。

企业内网爬虫只是互联网爬虫的一个小范围的应用,是爬虫技术与自动化技术的一种综合性应用,而且自动化技术的占比可能会比爬虫技术手段更多一些。



## 互联网爬虫

与企业爬虫相比，互联网爬虫就显得更为单一与常见，在这个数据唾手可得的年代，在数据中用爬虫淘金并不鲜见。如前面提及的搜索引擎本身就是“虫术大师”，只要是它们想爬的网站，几乎是没有爬不穿的。App Store 上最火的内容性 App 总是某些新闻类的聚合应用，大多数网站开发者都知道那只是一个聚合了各种新闻网站链接的综合性平台，它们的内容也是靠“放虫”才可能在各大新闻门户中获取第一手的新闻信息。更重要的是，这些新闻信息都是“免费”的，任何一个用户都可以轻易地从互联网上获取，这个用户当然也可以包括“虫子”。

互联网中存在大量如新闻资讯一类的免费内容，或是政府、企业、第三方机构、团体甚至个人共享的各种数据。例如，我们可以轻易地到气象局的网站上获取近十年某个地区的降雨量信息，或者从证券交易所获取当天各支股票的价格走势，又或者到微博上获知当天最具有传播性的某个事件的详情。换句话说，只要有清晰的目标数据源，只要你具有对数据源具有访问的权限，那么你也可以让爬虫为你代劳，一次性从数据源上获取所有你想要的数据。

要通过爬虫顺利地互联网中爬取数据，那么就地了解这些数据的特质，然后采取针对性手段才可能做到无往不利。一般来说，互联网中可爬取的数据可分为以下几种：

(1) 一般性的网页——符合 W3C 规范的网页都可视为一种半结构化的内容，可以通过一些页面元素分析工具从网页中读取指定数据，由于网页开发的自由度极大，几乎没有哪个网站的结构是完全相同的。而且可变因素也很多，可能网页读取要通过权限的审查，或者网页由客户端的 JavaScript 进行绘制才能呈现最终效果，甚至网页可能来源于 CDN，其内容未必是最新的，只是某个网络缓存的副本，等等。不过不用担心，当你完全掌握了虫术，这一切对你将不再是阻挡。

(2) API 资源——API 资源是最适合爬取的数据源（没有之一），因为 RESTful API 都是结构化数据，会以 XML 或者 JSON 的形式进行调用或者返回，这些数据内容即使没有 API 说明手册一般也能读懂。

(3) 文件资源——文件资源属于最麻烦的数据源了，除非爬取的文件是以结构化数据格式呈现的，否则作为自由文本，由于是非结构化的，我们需要对文本的内容进行一些后处理，要让爬虫“读懂”这些文本内容，再判断哪些内容是获取的目标。

(4) 媒体资源——如图片和视频等，其爬取的动作基本与文件类似，只是由于图片与视频等资源一般来说都比较大，可能还需要对文件的元信息进行一些分析以判断其是否具有爬取的价值，以避免让爬虫过多地消耗不必要的网络流量与爬取时间。

### 1.1.3 Python与爬虫

本书所讲述的虫术都是基于 Python 的，读者可能会产生这样的疑问：“不可以用其他语言



来编写爬虫吗？”当然，所有的语言都可以编写爬虫，Python 只是其中的一种实现手段，但 Python 绝对是其中的佼佼者。

首先，Python 具有极为简单的语法与跨平台的支持，Windows、Linux、macOS 都完美支持 Python，甚至还可以将 Python 放到如 Arduino YUN、树莓派等智能设备的上位机上运行，这是除 C 语言外其他语言都无法企及的。

其次，Python 具有极其庞大的第三方资源库。在国外，Python 早已成为一门教学级和工具级的语言，除了软件开发的专业领域，也应用到了其他各类科学计算领域，而且通过社区，源自于各个科学领域的专用的资源包都可被共享。

再者，Python 具有脚本级的简单性。得益于编译器的能力，Python 的源代码可被编译为 C 语言的原生码运行，也就是说，Python 具有脚本的简洁易懂的特性，同时具有 C 语言一般的高效运行的效能。

最后，由于各科学领域的贡献与社区的积累，Python 拥有大量在大数据与科学计算方面既稳定又出色的工具库。例如，热门的爬虫框架、简单的 Web 开发框架、各种高等数学的计算函数、自然语言及语义分析工具等，而且大数据应用领域内的工具绝大多数都会首先支持 Python 的编程接口，这一切使得 Python 在大数据应用开发上具有综合性的优势。

综上所述，使用 Python 这门具有强悍综合能力、广泛工具支持、入门简单的语言，可以让我们在虫术这条路上走得更快、更远。

## 1.1.4 Python 的网络爬虫框架

进入爬虫的世界后你会发现有两条路可以走，一条是“重新发明轮子”，一切都由你自己实现；另一条路则是使用别人的轮子，站人巨人的肩膀上看世界，站得高自然看得远。

为了学习底层实现，有必要从底层的源码入手，这样说并不代表我支持要走第一条路，我是个很懒的人，与其重新发明轮子不如在别人的基础上改良或者发展。当上层的功能堆积到一定程度时会自然组合成新的发明，因此我推荐使用已有的轮子，在别人的基础上发展，这样学得更快，而且更容易写出实用的东西。

### 主流框架

我们先从框架入手，看看现在 Python 世界中有哪些主流的网络爬虫框架。

#### ➤ PySpider

PySpider (<https://github.com/binux/pyspider>) 是国人做的一个爬虫架构的开源化实现，具有以下特性：

- Web 界面编写调试脚本，启停脚本，监控执行状态，查看活动历史，获取结果产出；

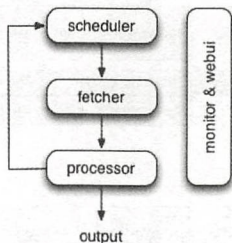
- 提供 SaaS 服务，可以在线提交部署；
- 支持 MySQL、MongoDB、SQLite；
- 原生支持抓取 JavaScript 的页面；
- 组件可替换，支持单机/分布式部署，支持 Docker 部署；
- 具有强大的调度控制；
- 灵活可扩展，稳定可监控。

这也是绝大多数 Python 爬虫的需求——定向抓取，结构化解析。但面对结构迥异的各种网站，单一的抓取模式并不一定能满足我们的要求，灵活的抓取控制是必需的。为了实现这个目的，单纯的配置文件往往不够灵活，于是，通过脚本去控制抓取是最后的选择。而去重调度、队列、抓取、异常处理、监控等功能作为框架，提供给抓取脚本，并保证灵活性。最后加上 Web 的编辑调试环境，以及 Web 任务监控，即组成了这套框架。

PySpider 的设计基础：以 Python 脚本驱动的抓取环模型爬虫。

- 通过 Python 脚本进行结构化信息的提取，follow 链接调度抓取控制，实现最大的灵活性。
- 通过 Web 化的脚本编写、调试环境。Web 展现调度状态，抓取环模型成熟稳定，模块间相互独立，通过消息队列连接，从单进程到多机分布式灵活拓展。

PySpider 的架构主要分为 scheduler（调度器）、fetcher（抓取器）和 processor（脚本执行），如下图所示。



- 各个组件间使用消息队列连接，除了 scheduler 是单点的，fetcher 和 processor 都是可以多实例分布式部署的。scheduler 负责整体的调度控制。
- 任务由 scheduler 发起调度，fetcher 抓取网页内容，processor 执行预先编写的 Python 脚本，输出结果或产生新的提链任务（发往 scheduler），形成闭环。
- 每个脚本可以灵活使用各种 Python 库对页面进行解析，使用框架 API 控制下一步抓取动作，通过设置回调控制解析动作。

PySpider 在最近几年的关注度颇高，主要得益其学习曲线比较平缓，代码的编写非常简单，而且容易理解，非常适合入门级和一些“短平快”的小型应用。



更高的易用性自然就会削弱其自定义的能力,在这方面 PySpider 的扩展能力是稍显不足的,而且对于千万级以上的增量式爬网,其去重(这两个概念在高阶虫术中会重点讲解)能力很弱,PySpider 更关注对数据入库的去重而缺失了对生成请求的 URL 的去重。而且其文档与资源相对其他老牌的爬虫框架要少,一旦出现问题,查找资料会显得困难重重。

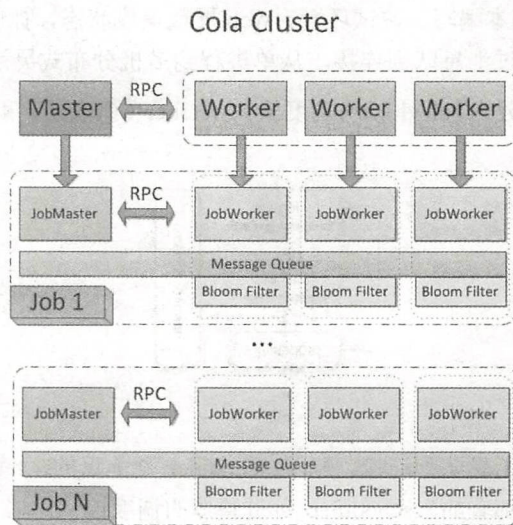
#### ➤ grab

grab (<http://docs.grablib.org/en/latest/#grab-spider-user-manual>) 是一个基于 pycurl/multicur 构建的网络爬虫框架。它是一个比较完善的爬虫框架,但相比于 PySpider,其学习曲线会显得陡峭一些,但工具与资源上的配置就与 PySpider 相差甚远。

#### ➤ Cola

Cola (<https://github.com/chineking/cola>) 是一个分布式的爬虫框架,用户只需编写几个特定的函数,而无须关注分布式运行的细节,任务会自动分配到多台机器上,整个过程对用户是透明的。

Cola 集群需要一个 Master 和若干个 Worker,每台机器只能启动一个 Worker。但是,集群不是必需的,在单机模式下也可以运行。Cola 集群集群如下图所示。



在 Cola 集群中,当一个任务被提交时,Cola Master 和 Worker 会分别启动 JobMaster 和 JobWorker。对于一个 Cola Job,当 JobWorker 启动完成后,会通知 JobMaster,JobMaster 等待所有 JobWorker 启动完成后开始运行 Job。在一个 Cola Job 启动时,会启动一个消息队列(Message Queue,主要操作是 put 和 get,Worker 抓取到的对象会被“put”到队列中,而要抓取新的对象时,只要从队列中获取即可),每个 JobWorker 上都存在消息队列节点,同时会有一个去重模块(bloom filter 实现)。

Cola 还不够稳定，目前处于持续改进的状态。而且 Cola 还没有在较大规模的集群上测试，但它也属于一个比较值得关注与学习的框架。

### ➤ Scrapy

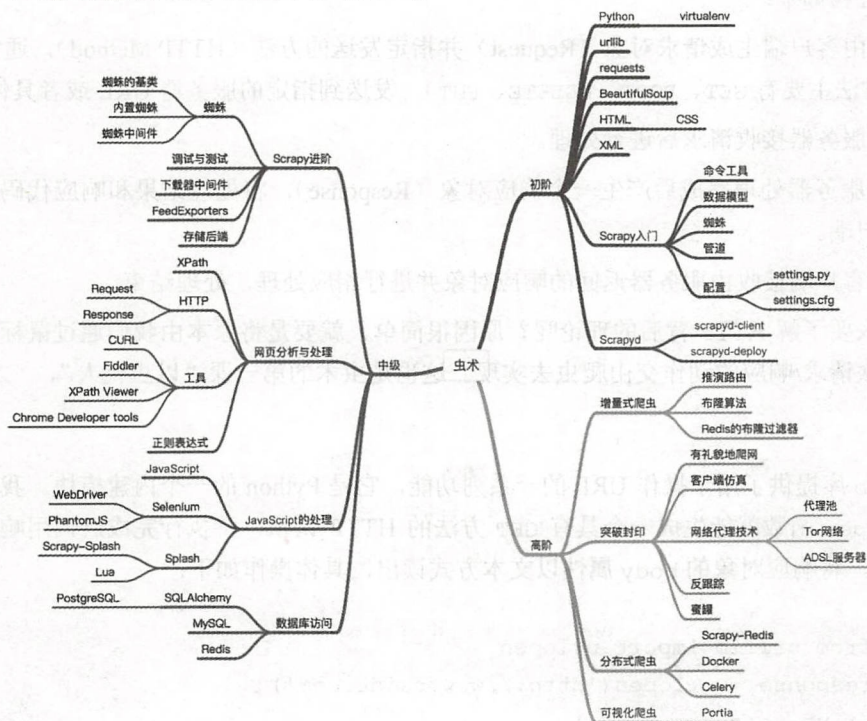
一个基于 twisted 开发的可能是 Python 世界中最出名也是使用者最多的爬虫框架。同时它也是本书的主角，既然它是主角，那么在本书中的分量一定不少，这里容许我先卖个关子，后面自然对它有详尽的介绍。

### 其他

- Portia——基于 Scrapy 的可视化爬虫。地址为 <https://github.com/scrapinghub/portia>。
- Restkit——Python 的 HTTP 资源工具包。它可以让你轻松地访问 HTTP 资源，并围绕它建立的对象。地址为 <https://github.com/benoitc/restkit>。

## 1.1.5 虫术技术路线图

虫术的各个阶段涉及的技术非常庞杂，为了让读者有一个全面的认识，我特意将三个阶段中所要学习与使用的技术归纳成下图以作参考。

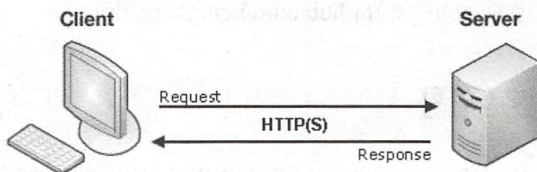




## 1.2 实例：简单的爬虫

我是一个实用主义者和实践主义者，尤其是一个极为疯狂的代码控，我喜欢在了解一门技术的知识脉络后就动手写一些小程序，做一些例子。我认为实践是另一种形式的思考，能比较容易地理解那些让人晦涩难懂的理论。毕竟代码本来就是一种语言，一种与计算无障碍沟通的语言，所以建议读者能换一种方式来阅读此书，先在你的机器上搭建一个 Python 环境，然后一边看以下内容，一边动手尝试，这样你可能会得到更多。

在开始之前，最好了解一些关于 HTTP 协议的知识，如果你已熟知它的一切，那么你可以跳过本段。无论你以前是否了解过 HTTP，至少你一定使用过它，因为浏览器就是依赖于它工作的。HTTP 协议的原理与工作流程非常简单，如下图所示。



具体过程如下：

- (1) 由客户端生成请求对象 (Request) 并指定发送的方法 (HTTP Method)，通常为 GET (HTTP 方法主要有 GET、POST、DELETE、PUT)，发送到指定的服务器 URL 或者具体 IP 上。
- (2) 服务器接收请求后进行处理。
- (3) 服务器处理完成后产生一个响应对象 (Response)，将处理结果和响应代码写入其中返回给客户端。
- (4) 客户端接收由服务器返回的响应对象并进行相应处理，处理结束。

为什么要了解 HTTP 背后的理论呢？原因很简单，就要是将原本由我们通过鼠标点击所完成的一次次请求/响应的动作交由爬虫去实现。这也是虫术的第一课“以虫代人”。

### urllib

urllib 库提供了用于操作 URL 的一系列功能，它是 Python 的一个内建模块。我们用它提供的 `urlopen` 函数就能生成一个具有 GET 方法的 HTTP 请求，待执行完成后调用响应对象的 `read` 方法，将响应对象的 `body` 属性以文本方式读出，具体操作如下：

```
>>> from urllib import urlopen
>>> response = urlopen('http://www.baidu.com')
>>> print response.read()
```



可以在命令行状态下直接输入 Python 指令, 进入 Python 的 shell 就可以直接输入以上代码运行并观察其执行的结果, 如下图所示。

```
RayOSX:~ Ray$ python
Python 2.7.10 (default, Feb 7 2017, 00:08:15)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from urllib import urlopen
>>> response = urlopen("http://www.baidu.com")
>>> print response.read()
<!DOCTYPE html>
<!--STATUS OK-->
```

```
<html>
<head>

<meta http-equiv="content-type" content="text/html;charset=utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=Edge">
<meta content="always" name="referrer">
<meta name="theme-color" content="#2932e1">
<link rel="shortcut icon" href="/favicon.ico" type="image/x-icon" />
<link rel="search" type="application/opensearchdescription+xml" href="/content-search.xml" title="百度搜索" />
<link rel="icon" sizes="any" mask href="//www.baidu.com/img/baidu.svg">


<link rel="dns-prefetch" href="//s1.bdstatic.com"/>
<link rel="dns-prefetch" href="//t1.baidu.com"/>
<link rel="dns-prefetch" href="//t2.baidu.com"/>
<link rel="dns-prefetch" href="//t3.baidu.com"/>
<link rel="dns-prefetch" href="//t10.baidu.com"/>
<link rel="dns-prefetch" href="//t11.baidu.com"/>
<link rel="dns-prefetch" href="//t12.baidu.com"/>
<link rel="dns-prefetch" href="//b1.bdstatic.com"/>


<title>百度一下，你就知道</title>


<style id="css_index" index="index" type="text/css">html,body{height:100%}
html{overflow-y:auto}
body{font:12px arial;text-align:center;background:#fff}
body,p,form,ul,li{margin:0;padding:0;list-style:none}
body,form,#fm{position:relative}
td{text-align:left}
img{border:0}
a{color:#00c}
```

是不是用代码来描述比用文字描述更加清楚呢? `read()` 方法将响应对象的 `body` 内容以文本方式读出并打印到屏幕上, 这些内容就是我们平时经常打开的百度网页上的原生代码内容了。

至此, 我们已经将这个爬虫写了一半, 剩下就是要从爬取回来的内容中提取需要的数据信息了。

## BeautifulSoup

BeautifulSoup (BS) 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库。它能够通过你喜欢的转换器实现常用的文档导航、查找、修改文档的方式。BeautifulSoup 会帮你节省数



小时甚至数天的工作时间。在前面返回的网页内容中,如果要从文字中直接查找某些内容是非常麻烦的,但如果使用 BeautifulSoup,就可以将整个文档内容载入并生成一系列的对象,这样就能像在浏览器上使用 jQuery 一般轻松地分析网页的内容。

由于 BeautifulSoup 是一个第三方库,所以需要用到 pip 命令将其安装到 Python 环境中:

```
$ pip install beautifulsoup4
```

由于 BeautifulSoup 需要 XML 解释器的支持,所以如果你的机器上没有安装这类解释器的库,在执行上述命令时会出现错误。可以运行 `$ pip install lxml` 来安装 lxml XML 解释器的支持。

如果是使用 macOS 的用户,则建议采用 homebrew 安装 lxml,这样会省去很多麻烦:

```
$ brew install lxml
```

这次我将代码写入一个名为 `myscript.py` 的 Python 源码文件中:

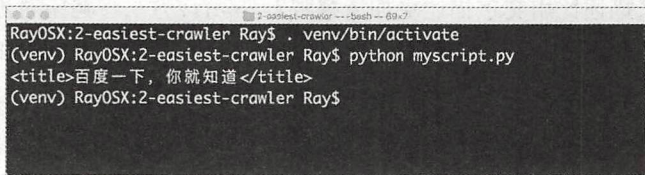
```
#!/bin/python
from urllib import urlopen
from bs4 import BeautifulSoup
response = urlopen('http://www.baidu.com')
bs = BeautifulSoup(response.read(), "html.parser")
print bs.title
```

上述代码就是导入一个 BeautifulSoup 对象并将响应的网页源代码内容作为其构造函数的初始参数,然后从 BeautifulSoup 的实例 bs 中获取当前网页上标题的文本内容。

在命令行用 Python 命令运行它:

```
$ python myscript.py
```

结果如下图所示。



```
RayOSX:2-easiest-crawler Ray$ . venv/bin/activate
(venv) RayOSX:2-easiest-crawler Ray$ python myscript.py
<title>百度一下, 你就知道 </title>
(venv) RayOSX:2-easiest-crawler Ray$
```

这就是我们的第一爬虫,一个只有 6 行代码就能完成的简单爬虫!“以虫代人”只是一个开始,这个例子旨在让读者能看到“虫”的本质,并试图让读者对其产生兴趣与足够的好奇心。

## 小结与思考

在进入下一节之前，希望读者能思考以下问题，可以带着这些问题在下文中找出答案。

- 数据越来越多时应如何处理？
- 如何让爬虫在规定的时间内自动运行？
- 如何让爬虫运行得更快？

## 附：采用VirtualEnv与Python的根环境隔离

在后面的章节中会用到越来越多的第三方包，每次执行 `pip` 或者 `easy_install` 命令都会将这些第三方包安装到 Python 的根目录下。当机器上的 Python 项目越来越多时，每个项目中采用的依赖包版本可能也不尽相同。一旦依赖包更新，很有可能会导致一些旧的包与源代码不能工作而产生“更新地狱”。

为了确保每个 Python 项目之间可以保持独立与隔离，让每个项目可以具有独立的依赖包环境，我们可以使用 `virtualenv`，它可以说是所有 Python 老手都会用到的一个重要的工具。

### ➤ 安装 `virtualenv`

```
$ easy_install virtualenv
```

`virtualenv` 实际上是从根目录将 Python 根目录复制到当前所建立的虚目录之中的，当调用 `activate` 指令激活 `virtualenv` 虚环境后，当前目录（`venv/bin/python`）就会被虚拟为 Python 的根目录，而不再指向系统的根（`/bin/python`），所有执行 `pip` 命令安装的依赖包都会被安装到当前的 Python 虚环境中。当安装 `virtualenv` 命令后，每建立一个 Python 项目都需要执行一次以下做法。

### ➤ 初始化与激活虚环境

```
$ mkdir scrapyng & $_  
scrapyng$ virtualenv venv  
scrapyng$ ./venv/activate
```

## 1.3 内容分析进阶

爬虫可以从内网或互联网上爬取一大堆的数据再保存到本地，这只是一种初阶的爬虫，它和一个垃圾下载器的区别几乎不大。因为这种爬虫系统很笨，除了会“吃”什么也不会，几乎不能用于实际的开发场景中，只能作为学习例子。爬虫系统中很重要的一步就是让虫子具有“分析能力”，要对爬到的内容按照我们给定的规则从内容中抽取数据，而不是选择直接保存下来。



这一点也在前文中使用 BeautifulSoup 时有所体验了。BeautifulSoup 能很优雅地将一段或者整个网页文本载入并实例化为各种对象属性, 这样我们就可以很方便地采用面向对象的方式来访问结构化后的网页文本内容了。除了有这么一个分析“神兵”辅助, 我们还得需要有扎实的 HTML 网页的编程基础。

在深入讲述 BeautifulSoup 一些相关内容之前, 需要对网络文本结构的一些基础知识进行介绍, 希望能给对此不甚了解的读者理解本书后面的内容带来帮助。

## HTML

在学习虫术的初始阶段, 我们还需要了解 Python 技术以外的相关知识, HTML 可谓是其中重要的一项。爬虫爬取的目标是互联网, 而网页则是一种基本的结果性的输出文本, 反过来说, 也就是爬虫的基本输入。所以要学好爬虫, 得先了解 HTML。可能对于初学者来说是这一种打击, “是否我现在就得停下来先学习完 HTML 再回到这里重新开始?” 当然, 如果想更扎实地掌握虫术, 我建议你这样做。但凡事也有捷径, 也可以“边做边学”, 以一种技术为主线, 触类旁通, 这也是我快速学习技术的一种方法。

在虫术中使用 HTML 并不像网页开发或网站开发那样非得对 HTML 所有的技术内容如臂使指般熟悉, 毕竟虫子只是文档的一个“读者”而不是“作者”, 只要读懂文档结构, 了解主要内容与思想足以。接下来介绍一个 5 分钟学习 HTML 的方法, 只要掌握以下几点就够了。

首先, 标准的 HTML 其实就相当于一份 XML, 尤其是 HTML5 标准。所以 HTML 就是一份以 html 为根元素的文档树, 每个元素都可以通过特定的路径被搜寻到。

其次, a 元素也就是网络链接是 HTML 最重要和最常见的元素, 当然这个元素对于我们的虫子也是相当重要的, 它可以说是我们实现自动“爬”取的重要线索。

再者, 其他元素都可以看作一般性的元素, 暂时不需要理会它们在网页呈现中会有什么作用, 只需要知道每个元素有属性 (attribute) 和文本 (text) 就够了。

最后谨记一点, 学习靠的不是速度而是积累, 我所提供的方法是极为粗线条的且仅适合本书后面讲述的内容所需要的基础技术, 对文本结构了解得越深, 对你将来在实际开发项目中应用虫术是具有很大增益的。最好的积累办法是, 遇到不了解的元素可以到 Firefox 的技术官网上查看元素的具体用法, 一点一滴地积累 HTML 的相关知识。

## XML

符合 HTML5 规范的 HTML 就是 XML 的一个子集, 这在前面已经提过了。所以要掌握 XML, 并且用 Python 的工具包访问其中的内容也就是几分钟的事情。XML 就是具有一个根元素的规范化的树状文本而已。每个元素都具有属性与文本, 就这么简单。

在爬取的网络资源中有非常多的资源是以 XML 格式呈现的, 这是由于 JSON 格式在没有

兴起以前（大约在 15 年前），互联网掀起过一次 XML 化的浪潮，大量的数据和 API 都以 XML 方式进行交互。经过多年的沉淀，网络上的 XML 俨然成为一种仅次于 HTML 的通用文本了。包括前文提及的新闻聚合就是一种以 XML 格式定义的网络供稿（RSS/ATOM），其中含有大量的链接信息，在某些场合下我们甚至可以将其作为一种进行大规模爬网的索引来使用！

XML 本身就是被严格结构化的，对于我们做文本分析是极具便利性的，只需要用一个 XML 分析包（lxml）就可以轻易地从某个指定的对象或者对象路径上搜寻到指定的元素及内容。

### 1.3.1 选择器

选择器是在结构化文档中用于定位一个或多个符合条件元素的一种语法，当然这种语法也需要相应的软件工具包支持。

#### CSS选择器

相信对 HTML 有所了解的读者都知道 CSS 就是 HTML 的样式表，负责改变 HTML 文档元素呈现的外观样式。而 CSS 对于编程来说也是非常有用的，因为 CSS 在被应用到 HTML 文档上的某类或某个元素时都是需要指定的。在 HTML 的具体元素内用 class 属性进行指定是一个依赖性很强的方案，而另一种办法就是对一般性的元素进行指定。例如，我们要网页上的所有链接都显示为红色，那么 CSS 就要这样写：

```
a {  
    color:red;  
}
```

此时 a 就是一个样式选择器，指定所有文档内的链接元素。如果只是指定网页内的表单中的链接才具有红色这一特性，那么就可以给 a 加上 form 元素的限定：

```
form a {  
    color:red;  
}
```

CSS 选择器的基本语言就是容器元素在前、子元素在后，其实也是一种树状的路径的指定方式。由于 CSS 选择器的语法内容非常多，受篇幅所限，不会在此一一赘述，如果有需要，可以访问 [https://developer.mozilla.org/zh-CN/docs/Learn/CSS/Introduction\\_to\\_CSS/Selectors](https://developer.mozilla.org/zh-CN/docs/Learn/CSS/Introduction_to_CSS/Selectors) 了解 CSS 选择器的语法内容。

那为什么要学习 CSS 选择器呢？因为这是一种最简单的文本元素定位的方法，我们要在虫



子中定位某个具有特定性质的元素（例如，使用 BeautifulSoup）就得采用 CSS 选择器。

### XPath

XPath 是对 XML 的一种路径选择法，在 HTML 文档上，XPath 使用起来比 CSS 选择器更复杂，但却能得到更好的性能，在很多大规模爬网的场合是非常适用的。而且除了 HTML，它还能应用于 XML 文档，这是 CSS 选择器不具备的。

由于 XPath 的语法相对复杂，可以访问 <https://developer.mozilla.org/zh-CN/docs/Web/XPath> 来了解它的具体技术细节，在后文中会对具体的代码例子进行细致的讲述。

## 1.3.2 深入 BeautifulSoup

有了以上的预备知识，我们可以更深入地了解 BeautifulSoup 的具体用法。BeautifulSoup 的具体内容有不少，在实际使用中我们并不会使用到所有内容，所以本节只介绍最常用的一部分，其他部分可以参见它的详细技术文档（在本节最后有具体参考文档的地址）。

Beautiful Soup 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库。它能够通过你喜欢的转换器实现惯用的文档导航、查找、修改文档的方式。执行以下命令就能直接安装到 Python 环境中：

```
$ pip install beautifulsoup
```

### BeautifulSoup 中的对象

Beautiful Soup 将复杂的 HTML 文档转换成一个复杂的树形结构，每个节点都是 Python 对象，所有对象可以归纳为 4 种：Tag、NavigableString、BeautifulSoup 和 Comment。

#### ➤ BeautifulSoup 对象

这是 BeautifulSoup 程序入口对象，要使用 BeautifulSoup 就需要先实例此对象，BeautifulSoup 的构造函数接受三种类型的字符串作为初始参数，第一方式是使用 HTML 代码片，具体如下所示。

```
from bs4 import BeautifulSoup
html = BeautifulSoup("<h1>这是一份测试文档<h1>", "html.parser")
print html.h1
```

这个例子在本书的第一个示例中就已经讲述过了，有了 BeautifulSoup 实例之后，其他相关的对象会在它被实例化时根据输入的 HTML 文档一一被构造，并可通过属性引用其构造的相关对象属性访问文档内容。

第二种方式是输入完整的 HTML 文档, 比如前面用 `openurl` 函数将响应结果作为其初始参数:

```
from bs4 import BeautifulSoup
from urllib import urlopen
html = BeautifulSoup(urlopen("http://www.baidu.com"), "html.parser")
print html.h1
```

第三种方式是直接向 `BeautifulSoup` 输入指定的网页文件, `BeautifulSoup` 会打开这个网页文件的内容并载入 `BeautifulSoup` 实例中。

```
from bs4 import BeautifulSoup
html = BeautifulSoup("navtive.html", "html.parser")
print html.h1
```

### ➤ Tag对象属性

Tag 对象是 XML 或 HTML 原生文档中的元素标签对象, 当 `BeautifulSoup` 实例化时, 本质上就是将输入的字符串内容生成树状层级关系结构的 tag 集合。所以当我们采用 `BeautifulSoup` 作为文档内容分析工具时, 用得最多的就是这个 tag 对象。接下来就举例说明一下它的用法:

```
soup = BeautifulSoup('<b class="boldest">这是一个加粗的文本</b>')
tag = soup.b
type(tag)
# <class 'bs4.element.Tag'>
```

Tag 对象有很多方法和属性, 其中最重要的属性是 `name` 和 `attributes`。

#### name 属性

每个 tag 都有自己的名字, 可以通过 `.name` 属性来获取, 主要用来判断当前的标签实例具体的标签名是什么:

```
>>> print tag.name
# u'b'
```

如果改变了 tag 的 `name`, 则影响所有通过当前 `Beautiful Soup` 对象生成的 HTML 文档:

```
>>> tag.name = "blockquote"
>>> print tag
```



```
# <blockquote class="boldest">Extremely bold</blockquote>
```

attrs 属性字典

一个 tag 可能有很多个属性。<b class="boldest">有一个“class”的属性, 值为“boldest”。tag 的属性的操作方法与字典相同, 或者说 tag 对象实例本身就是一个元素标签的属性字典:

```
>>> print tag['class']
# u'boldest'
```

也可以直接通过 attrs 取得这个属性字典:

```
>>> print tag.attrs
# {u'class': u'boldest'}
```

tag 的属性可以被添加、删除或修改。再说一次, tag 的属性操作方法与字典一样:

```
>>> tag['class'] = 'verybold'
>>> tag['id'] = 1
>>> print tag
# <blockquote class="verybold" id="1">这是极粗文本</blockquote>
>>> del tag['class']
>>> del tag['id']
>>> print tag
# <blockquote>这是极粗文本</blockquote>
>>> print tag['class']
# KeyError: 'class'
>>> print(tag.get('class'))
# None
```

我们在虫术中采用 tag 对象读取属性值的频次会比修改标签内容更高, 甚至基本上不用 BS 来修改标签内容, 所以在此也不作过多的解释, 仅仅作为一段扩展阅读。

### 1.3.3 元素的搜寻

前面介绍了一大堆 BeautifulSoup 的相关内容, 本节才真正是使用 BeautifulSoup 的意义所在, 这就是 BeautifulSoup 的元素搜索功能, 这个功能也是为什么 BS 的性能不高却依然能在 Python 的生态圈中占有一席之地原因。

BeautifulSoup 提供了一系列的 `find()` 方法,即使内容结构再复杂,也能简单、快速地定位到目标元素上。

我们从一个更具体的例子来入手,以下是一个关于 iPhone 手机产品的简单产品列表的网页。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>产品列表</title>
</head>
<body>
<div>
  <ul class="products">
    <li id="iphone5s-64-gray">
      <article>
        <header>iPhone 5s</header>
        <p class="color">
          <span>颜色</span><span>深空灰</span>
        </p>
        <p class="flash">
          <span>内存</span><span>64G</span>
        </p>
        <p class="price">
          <span>价格</span><span>1080</span>
        </p>
      </article>
    </li>
    <li id="iphone5s-128-b">
      <article>
        <header>iPhone 5s</header>
        <p class="color">
          <span>颜色</span><span>黑</span>
        </p>
        <p class="flash">
          <span>内存</span><span>128G</span>
        </p>
        <p class="price">
```



```

        <span>价格</span><span>1280</span>
    </p>
</article>
</li>
<li id="iphone7-128-b">
    <article>
        <header>iPhone 7</header>
        <p class="color">
            <span>颜色</span><span>磨砂黑</span>
        </p>
        <p class="flash">
            <span>内存</span><span>128G</span>
        </p>
        <p class="price">
            <span>价格</span><span>5688</span>
        </p>
    </article>
</li>
</ul>
</div>
</body>
</html>

```

假如我们要从这个网页中提取出所有产品的名称和价格，那么可以通过 BeautifulSoup 提供的 find 方法实现。首先，对网页的结构进行分析：产品是以<li>元素为容器，<header>元素的内容是产品的名称，在其之下的每个<p>元素内则是具体的产品特性名称和参数值，每个<p>元素又带有独特的 class 属性标识其具体的数据意义。根据本示例的要求，我们只需要提取所有的<header>元素，以及带有 class="price"类标识的<p>元素中的第二个<span>内的数据即可。

具体实现代码如下所示。

```

from bs4 import BeautifulSoup

html = BeautifulSoup(open('index.html').read(), 'lxml')
products = []

```

```

for ele in html.find_all("article"):
    _name = ele.header.text
    _value = int(ele.find("p", class_="price").find("span").find_next("span").text)
    products.append((_name, _value))

print products

```

接下来详细解释一下以上代码。首先采用 `find_all` 方法从文档中找出所有的<article>元素，在循环体内可以获得具体的 article 的 Tag 实例，那么可以直接通过名称来引用属于它的子元素，因此从 `ele.header.text` 中就可以得到产品的名称。其次，用 `find` 方法在 article 拥有的三个<p>元素中选出具有 `class="color"` 的一个，分别再选定第一个<span>元素作为相对定位的参考，最后用 `find_next` 找到与<span>相邻的具有价格值的<span>元素。

BeautifulSoup 提供的这一系列的 `find` 方法可以让我们以对象的方式来过滤和筛选文档树的节点，这种方式的优点是学习成本低，而且代码易读性强。

`find()` 系列方法参考如下表所示。

方 法	说 明
<code>find()</code>	查找返回文档中符合条件的第一个元素
<code>find('p')</code>	按照指定的元素标签名称查找元素并返回第一个符合条件的元素
<code>find(text="newtext")</code>	查找并返回具有指定文本内容的第一个元素
<code>find(attrs={'id': 'value'})</code>	查找并返回与指定属性值匹配的的第一个元素
<code>find(class_='value')</code>	查找具有指定样式类的第一个匹配元素
<code>find_all()</code>	查找并返回所有元素
<code>find_parent()</code>	返回父元素
<code>find_parents()</code>	返回所有的祖先元素
<code>find_sibling()</code>	返回第一个相邻的元素
<code>find_siblings()</code>	返回所有的相邻的元素
<code>find_next()</code>	从当前元素开始查找返回下一个符合条件的元素
<code>find_all_next()</code>	从当前元素开始查找返回下一个符合条件的的所有元素
<code>find_previous()</code>	从当前元素开始查找返回上一个符合条件的元素
<code>find_all_previous()</code>	从当前元素开始查找返回下一个符合条件的的所有元素

`find_all_*`方法与 `find()`方法具有相同的参数。



看完以上的逻辑解释是不是觉得非常费劲? 其实我更喜欢采用 CSS 选择器作为元素的筛选条件。

### CSS选择器

BeautifulSoup 和 Tag 对象提供了两个支持 CSS 选择器的方法, 分别是:

- `select(css_selector)` —— 返回所有匹配指定 CSS 选择器的元素。
- `select_one(css_selector)` —— 只返回一个与指定 CSS 选择器匹配的元素。

我们用 CSS 选择器来改写上例:

```
from bs4 import BeautifulSoup

html = BeautifulSoup(open('index.html').read(), 'lxml')
products = []

for ele in html.select("article"):
    _name = ele.header.text
    _value = int(ele.select_one('p.price > span:nth-of-type(2)').text)
    products.append((_name, _value))

print products
```

用 CSS 选择器是否会显得更清晰明了呢?

## 1.3.4 乱码与中文编码

我们在对爬虫爬取回来的结果进行内容检索时, 尤其是带有中文的内容, 经常会出现各种乱码的情况, 这是中文软件生态中不可回避的问题。相信不少程序员都碰到过这个问题, 尤其在 Python 中, 对文字的编码有很多种方式, 从源代码文本到运行期变量都会出现文字编码的乱码现象。但这并不是 Python 导致的, 很可能是因为我们所选用的编码方式不正确所导致的。接下来就从根源说起, 先来看看文本编码的一些背景知识, 然后回到 Python, 学习如何处理中文的文本编码。

### 文本编码

ASCII 码 (American Standard Code for Information Interchange, 美国信息交换标准码) 是目前计算机中使用最广泛的字符集及其编码, 由美国国家标准局 (ANSI) 制定。它已被国际标准

化组织 (ISO) 定为国际标准, 称为 ISO 646 标准。ASCII 字符集由控制字符和图形字符组成。

在计算机的存储单元中, 一个 ASCII 码值占一个字节 (8 个二进制位), 其最高位 (b7) 用作奇偶校验。所谓奇偶校验, 是指在代码传送过程中用来检验是否出现错误的一种方法, 一般分奇校验和偶校验两种。奇校验规定: 正确的代码一个字节中 1 的个数必须是奇数, 若非奇数, 则在最高位 b7 添 1。偶校验规定: 正确的代码一个字节中 1 的个数必须是偶数, 若非偶数, 则在最高位 b7 添 1。ISO 8859 的全称为 ISO/IEC 8859, 是国际标准化组织及国际电工委员会 (IEC) 联合制定的一系列 8 位字符集的标准, 已经定义了 15 个字符集。

Unicode 是一种在计算机上使用的字符编码。它是 <http://www.unicode.org> 制定的编码机制, 要将全世界的常用文字都囊括进去。Unicode 为每种语言中的每个字符设定了统一且唯一的二进制编码, 以满足跨语言、跨平台进行文本转换和处理的要求。1990 年开始研发, 1994 年正式公布。随着计算机计算能力的增强, Unicode 得到了普及。

从 Unicode 2.0 开始, Unicode 采用了与 ISO 10646-1 相同的字库和字码, ISO 也承诺 ISO 10646 将不会给超出 0x10FFFF 的 UCS-4 编码赋值, 使得两者保持一致。

Unicode 的编码方式与 ISO 10646 的通用字符集概念相对应, 目前的实际应用的 Unicode 版本对应 UCS-2, 使用 16 位的编码空间。也就是每个字符占用 2 个字节, 基本满足各种语言的使用。实际上, 目前版本的 Unicode 尚未填满 16 位编码空间, 保留了大量空间作为特殊使用或将来扩展。

一个字符的 Unicode 编码是确定的, 但是在实际传输过程中, 由于不同系统平台的设计不一致, 以及出于节省空间的目的, 对 Unicode 编码的实现方式有所不同。Unicode 的实现方式称为 Unicode 转换格式 (Unicode Translation Format, UTF)。UTF 的两种实现方式如下所示。

- UTF-8: 8 位变长编码, 对于大多数常用字符集 (ASCII 中 0~127 字符), 它只使用单字节, 而对其他常用字符 (特别是中文、日文、韩文等象形文字), 它使用 3 字节。
- UTF-16: 16 位编码, 是变长码, 大致相当于 20 位编码, 值在 0 到 0x10FFFF 之间, 基本上就是 Unicode 编码的实现, 与 CPU 字序有关。

汉字编码有如下 4 种。

- GB2312 字集是简体字集, 全称为 GB2312(80)字集, 包括国标简体汉字 6763 个。
- BIG5 字集是中国台湾地区的繁体字集, 包括国标繁体汉字 13053 个。
- GBK 字集是简繁体字集, 包括 GB 字集、BIG5 字集和一些符号, 共 21003 个字符。
- GB18030 是国家制定的一个强制性大字符集标准, 全称为 GB18030-2000, 它的推出使汉字集有了一个“大一统”的标准。



**BOM**——Unicode 规范中推荐的标记字节顺序的方法是 BOM (Byte Order Mark)。在 UCS 编码中有一个叫“ZERO WIDTH NO-BREAK SPACE”的标记，它的编码是 FEFF。而 FFFE 在 UCS 中是不存在的字符，所以不应该出现在实际数据中。UCS 规范建议在传输字节流前，先传输标记“ZERO WIDTH NO-BREAK SPACE”。如果接收者收到 FEFF，就表明这个字节流是 Big-Endian 的；如果收到 FFFE，就表明这个字节流是 Little-Endian 的。因此标记“ZERO WIDTH NO-BREAK SPACE”又被称为 BOM。Windows 就是使用 BOM 来标记文本文件的编码方式的。

现在，梳理 ASCII 编码和 Unicode 编码的区别：ASCII 编码是 1 个字节，而 Unicode 编码通常是 2 个字节。

- 字母 A 用 ASCII 编码是十进制的 65、二进制的 01000001；
- 字符 0 用 ASCII 编码是十进制的 48、二进制的 00110000，注意字符'0'和整数 0 是不同的；
- 汉字“中”已经超出了 ASCII 编码的范围，用 Unicode 编码是十进制的 20013、二进制的 01001110 00101101。

你可以猜测，如果把 ASCII 编码的 A 用 Unicode 编码，则只需要在前面补 0 就可以，因此，A 的 Unicode 编码是 00000000 01000001。

新的问题又出现了：如果统一成 Unicode 编码，则乱码问题从此消失了。但是，如果你写的文本基本上全部是英文，则用 Unicode 编码比 ASCII 编码需要多一倍的存储空间，在存储和传输上十分不划算。

所以，本着节约的精神，又出现了把 Unicode 编码转化为“可变长编码”的 UTF-8 编码。UTF-8 编码把一个 Unicode 字符根据不同的数字大小编码成 1~6 个字节，常用的英文字母被编码成 1 个字节，汉字通常是 3 个字节，只有很生僻的字符才会被编码成 4~6 个字节。如果要传输的文本包含大量英文字符，则用 UTF-8 编码就能节省空间，如下表所示。

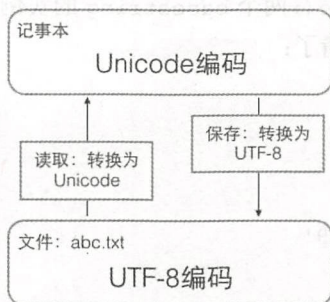
字 符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	x	01001110 00101101	11100100 10111000 10101101

从上面的表格还可以发现，UTF-8 编码有一个额外的好处，就是 ASCII 编码实际上可以看作 UTF-8 编码的一部分。所以，大量只支持 ASCII 编码的历史遗留软件可以在 UTF-8 编码下继续工作。

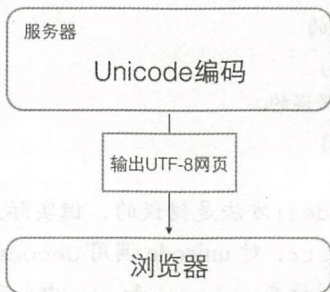
搞清楚了 ASCII、Unicode 和 UTF-8 的关系，我们就可以总结一下现在计算机系统通用的

字符编码工作方式：

- 在计算机内存中，统一使用 Unicode 编码，当需要保存到硬盘或者需要传输时，就转换为 UTF-8 编码。
- 用记事本编辑时，从文件读取的 UTF-8 字符被转换为 Unicode 字符保存到内存里，编辑完成后，再把 Unicode 转换为 UTF-8 保存到文件，如下图所示。



浏览网页时，服务器会把动态生成的 Unicode 内容转换为 UTF-8 再传输到浏览器，如下图所示。



所以很多网页的源码上会有类似<meta charset="UTF-8" />的信息，表示该网页用的正是 UTF-8 编码。

## Python的编码

Python 的字符串和文本编码问题经常会难倒不少的初学者，甚至“老手”也可能掉到坑里。而且 Python 2.7.x 与 Python 3 的编码又天差地别，更是让人头疼。Python 3 为了解决 Python 2 以前版本的编码问题，已对字符串进行统一的 Unicode 编码，此处暂且不表。由于多年来 Python 2.7.x 的用户比 Python 3 的用户要多得多，因此在此必须对 Python 2.7.x 的编码问题进行解释，以便解决以后在爬虫处理字符串时可能出现的编码乱象。



### ➤ str和unicode

str 和 unicode 都是 basestring 的子类。严格意义上说, str 其实是字节串, 它是 Unicode 经过编码后的字节组成的序列。对 UTF-8 编码的 str '汉' 使用 len() 函数时, 结果是 3。实际上, UTF-8 编码的 '汉' == '\xE6\xB1\x89'。而 unicode 才是真正意义上的字符串, 对字节串 str 使用正确的字符编码进行解码后获得, 并且 len(u'汉') == 1。

再看看 encode() 和 decode() 两个 basestring 的实例方法, 理解了 str 和 unicode 的区别后, 这两个方法就不会再混淆了:

```
# coding: UTF-8

u = u'汉'
print repr(u) # u'\u6c49'
s = u.encode('UTF-8')
print repr(s) # '\xe6\xba\x89'
u2 = s.decode('UTF-8')
print repr(u2) # u'\u6c49'

# 对 unicode 进行解码是错误的
# s2 = u.decode('UTF-8')
# 同样, 对 str 进行编码也是错误的。
# u2 = s.encode('UTF-8')
```

**注意:** 虽然对 str 调用 encode() 方法是错误的, 但实际上 Python 不会抛出异常, 而是返回另外一个相同内容但不同 id 的 str; 对 unicode 调用 decode() 方法也是这样。很不理解为什么不把 encode() 和 decode() 分别放在 unicode 和 str 中, 而是都放在 basestring 中, 但既然已经这样了, 我们就小心避免犯错吧。

### ➤ 字符编码声明

在源代码文件中, 如果用到非 ASCII 字符, 则需要在文件头部进行字符编码的声明:

```
#-*- coding: UTF-8 -*-
```

实际上 Python 只检查 #、coding 和 编码字符串, 其他的字符都是为了美观加上的。另外, Python 中可用的字符编码有很多, 并且还有许多别名, 还不区分大小写, 比如 UTF-8 可以写成 u8 (参见 <http://docs.python.org/library/codecs.html#standard-encodings>)。

另外需要注意的是, 声明的编码必须与文件实际保存时用的编码一致, 否则很容易出现代

码解析异常。现在的 IDE 一般会处理这种情况，改变声明后同时换成声明的编码保存，但文本编辑器控们需要小心。

### ➤ 读写文件

用内置的 `open()` 方法打开文件时，`read()` 读取的是 `str`，读取后需要使用正确的编码格式进行 `decode()` 操作。`write()` 写入时，如果参数是 `unicode`，则需要使用你希望写入的编码进行 `encode()` 操作，如果是其他编码格式的 `str`，则需要先用该 `str` 的编码进行 `decode()` 操作，转成 `Unicode` 后再使用写入的编码进行 `encode()` 操作。如果直接将 `unicode` 作为参数传入 `write()` 方法，则 Python 将先使用源代码文件声明的字符编码进行编码，然后写入。

```
# coding: UTF-8

f = open('test.txt')
s = f.read()
f.close()
print type(s) # <type 'str'>
# 已知是 GBK 编码，解码成 unicode
u = s.decode('GBK')

f = open('test.txt', 'w')
# 编码成 UTF-8 编码的 str
s = u.encode('UTF-8')
f.write(s)
f.close()
```

另外，模块 `codecs` 提供了一个 `open()` 方法，可以指定一个编码来打开文件，使用这个方法打开的文件读取返回的将是 `unicode`。写入时，如果参数是 `unicode`，则使用 `open()` 时指定的编码进行编码后写入；如果是 `str`，则先根据源代码文件声明的字符编码，解码成 `Unicode` 后再进行之前的操作。相对内置的 `open()` 来说，这个方法比较不容易在编码上出现问题。

```
# coding: GBK

import codecs

f = codecs.open('test.txt', encoding='UTF-8')
u = f.read()
f.close()
```



```
print type(u) # <type 'unicode'>

f = codecs.open('test.txt', 'a', encoding='UTF-8')
# 写入 Unicode
f.write(u)

# 写入 str, 自动进行解码编码操作
# GBK 编码的 str
s = '汉'
print repr(s) # '\xba\xba'
# 这里会先将 GBK 编码的 str 解码为 Unicode 再编码为 UTF-8 写入
f.write(s)
f.close()
```

### 与编码相关的方法

sys/locale 模块中提供了一些获取当前环境下的默认编码的方法。

```
# coding:gbk

import sys
import locale

def p(f):
    print '%s.%s(): %s' % (f.__module__, f.__name__, f())

# 返回当前系统所使用的默认字符编码
p(sys.getdefaultencoding)

# 返回用于转换 Unicode 文件名至系统文件名所使用的编码
p(sys.getfilesystemencoding)

# 获取默认的区域设置并返回元组 (语言、编码)
p(locale.getdefaultlocale)

# 返回用户设定的文本数据编码
# 文档提到 this function only returns a guess
p(locale.getpreferredencoding)
```

```
# \xba\xba 是'汉'的 GBK 编码
# mbcs 是不推荐使用的编码, 这里仅用于测试表明为什么不应该用
print r'''\xba\xba'.decode('mbcs'):', repr('\xba\xba'.decode('mbcs'))

# 在 Windows 上的结果(区域设置为中文(简体, 中国))
# sys.getdefaultencoding(): gbk
# sys.getfilesystemencoding(): mbcs
# locale.getdefaultlocale(): ('zh_CN', 'cp936')
# locale.getpreferredencoding(): cp936
# '\xba\xba'.decode('mbcs'): u'\u6c49'
```

### ➤ 使用建议

- 使用字符编码声明, 并且同一工程中的所有源代码文件使用相同的字符编码声明。
- 抛弃 `str`, 全部使用 `unicode`。按引号前先按一下 `u`, 最初做起来确实很不习惯而且经常会忘记再返回去补, 但这么做可以减少 90% 的编码问题。如果编码困扰不严重, 则可以不参考此条。
- 使用 `codecs.open()` 替代内置的 `open()`。
- 绝对需要避免使用的字符编码: MBCS/DBCS 和 UTF-16。这里说的 MBCS 不是指 GBK 什么的都不能用, 而是不要使用 Python 里名为 MBCS 的编码, 除非程序完全不移植。Python 中编码 MBCS 与 DBCS 是同义词, 指当前 Windows 环境中 MBCS 指代的编码。Linux 的 Python 实现中没有这种编码, 所以一旦移植到 Linux, 一定会出现异常! 另外, 只要设定的 Windows 系统区域不同, MBCS 指代的编码也是不一样的。

### UnicodeDammit

BeautifulSoup 工具包提供了一非常好的编码工具, 可以用更简单的方式来处理编码, 这就是编码自动检测 (UnicodeDammit)。它可以在 BeautifulSoup 以外使用, 在检测某段未知编码时可以使用这个方法:

```
>>> from bs4 import UnicodeDammit
>>> dammit = UnicodeDammit(u"\u4e2d\u56fd\u5c06\u8c03\u6574\u90e8\u5206\u6d88\u8d39\u54c1\u8fdb\u53e3\u5173\u7a0e\u67081\u65e5\u8d77\u5b9e\u65bd")
>>> print(dammit.unicode_markup)
中国将调整部分消费品进口关税 12 月 1 日起实施
```

如果在 Python 中安装了 `chardet` 或 `cchardet`, 那么编码检测功能的准确率将大大提高。输入



的字符越多, 检测结果越精确。如果事先猜测到一些可能的编码, 那么可以将猜测的编码作为参数, 这样将优先检测这些编码:

```
dammit = UnicodeDammit("Sacré bleu!", ["latin-1", "iso-8859-1"])
print(dammit.unicode_markup)
# Sacré bleu!
dammit.original_encoding
# 'latin-1'
```

使用 Unicode 时, Beautiful Soup 还会智能地把引号转换成 HTML 或 XML 中的特殊字符:

```
markup = b"<p>I just \x93love\x94 Microsoft Word\x92s smart quotes</p>"

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="html").unicode_markup
# u'<p>I just &ldquo;love&rdquo; Microsoft Word&rsquo;s smart quotes</p>'

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="xml").unicode_markup
# u'<p>I just &#x201C;love&#x201D; Microsoft Word&#x2019;s smart quotes</p>'
```

也可以把引号转换为 ASCII 码:

```
UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="ascii").unicode_markup
# u'<p>I just "love" Microsoft Word\'s smart quotes</p>'
```

这是很有用的功能, 但 Beautiful Soup 没有使用这种方式。默认情况下, Beautiful Soup 把引号转换成 Unicode:

```
UnicodeDammit(markup, ["windows-1252"]).unicode_markup
# u'<p>I just \u201clove\u201d Microsoft Word\u2019s smart quotes</p>'
```

有时文档的大部分编码都是用 UTF-8, 但同时包含了 Windows-1252 编码的字符, 就像微软的智能引号一样。一些包含多个信息来源的网站容易出现这种情况。UnicodeDammit.detwingle() 方法可以把这类文档转换成纯 UTF-8 编码格式, 看一个简单的例子:

```
snowmen = (u"\N{SNOWMAN}" * 3)
quote = (u"\N{LEFT DOUBLE QUOTATION MARK}I like snowmen!\N{RIGHT DOUBLE QUOTATION MARK}")
doc = snowmen.encode("utf8") + quote.encode("windows_1252")
```



```
print(doc)
# I like snowmen!

print(doc.decode("windows-1252"))
# â~fâ~fâ~f"I like snowmen!"
```

```
new_doc = UnicodeDammit.detwingle(doc)
print(new_doc.decode("utf8"))
# 🌨️🌨️🌨️"I like snowmen!"
```

在创建 BeautifulSoup 或 UnicodeDammit 对象前一定要先对文档调用 UnicodeDammit.detwingle(), 确保文档的编码方式正确。如果尝试去解析一段包含 Windows-1252 编码的 UTF-8 文档, 则会得到一堆乱码。比如 `â~fâ~fâ~f`“I like snowmen!”。

- BeautifulSoup 中文文档: [https://beautifulsoup.readthedocs.io/zh\\_CN/v4.4.0/](https://beautifulsoup.readthedocs.io/zh_CN/v4.4.0/)。
- XPath 在线文档: <https://developer.mozilla.org/zh-CN/docs/Web/XPath>。
- CSS 选择器: [https://developer.mozilla.org/zh-CN/docs/Learn/CSS/Introduction\\_to\\_CSS/Selectors](https://developer.mozilla.org/zh-CN/docs/Learn/CSS/Introduction_to_CSS/Selectors)。

正所谓实践见真知，所以本章的最后一节会用一个完整的示例涵盖本章提到过的理论，并且会将这个示例延用至后面的几章中。用不同的技术解决相同的问题就会有所对比，而且有温故知新的效果。

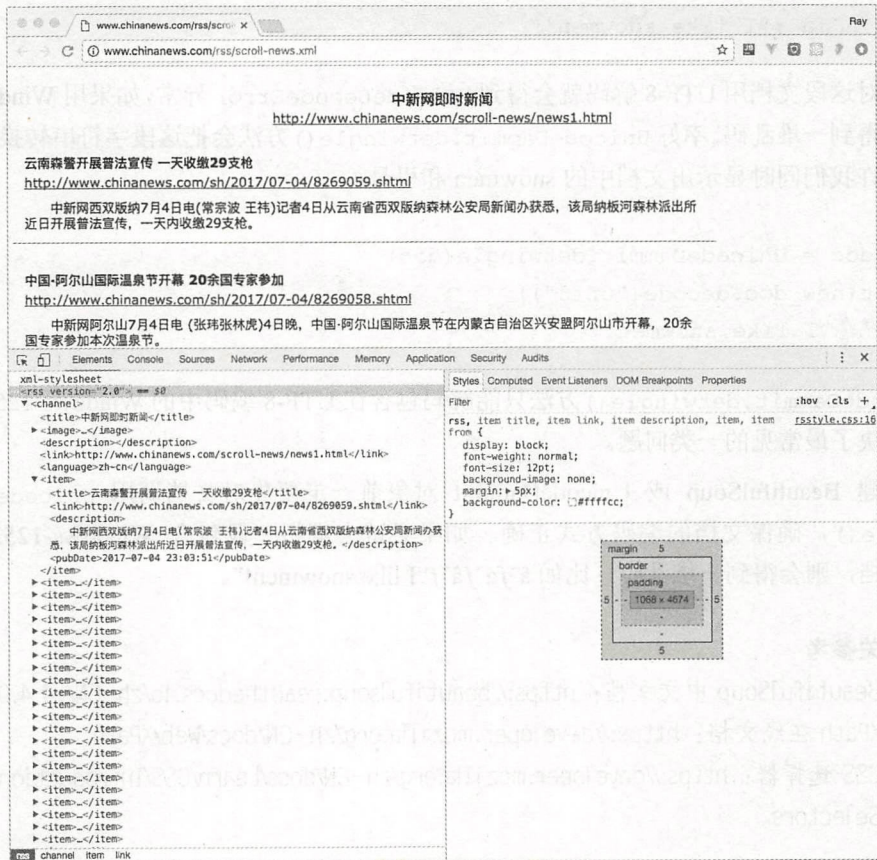
我一直强调当下的数据是唾手可得的，那么我们就来实现一个爬虫，从中国新闻网提供的





免费数据中爬取当天的新闻头条。这是一个很有代表性的例子，新闻供稿是一种特殊的 XML 格式，在现代互联网尤其是信息门户类网站、个人博客中尤为常见。用技术术语来讲就是 **RSS** 或者 **ATOM**，这是两种已经由 W3C 公布标准定义的数据供稿格式协议，而 **RSS** 则是国内最常用的一种协议。

我们可以到中新网打开即时新闻供稿来看看数据展示效果，然后用 Chrome 的开发者工具查看这个网页的结构，如下图所示。



以下是从中新网的即时新闻供稿截取下来的部分 RSS 源码：

```
<rss version="2.0">
  <channel>
    <title>中新网即时新闻</title>
    <image>
      <title>中新网即时新闻</title>
```



```

        <link>http://www.chinanews.com/scroll-news/news1.html</link>
        <url>http://www.chinanews.com/images/images1/logo2.gif</url>
    </image>
    <description/>
    <link>http://www.chinanews.com/scroll-news/news1.html</link>
    <language>zh-cn</language>
    <item>
        <title>云南森警开展普法宣传 一天收缴 29 支枪</title>
        <link>http://www.chinanews.com/sh/2017/07-04/8269059.shtml</link>
        <description>
            中新网西双版纳 7 月 4 日电 (常宗波 王玮) 记者 4 日从云南省西双版纳森林
            公安局新闻办获悉, 该局纳板河森林派出所近日开展普法宣传, 一天内收缴 29 支枪。
        </description>
        <pubDate>2017-07-04 23:03:51</pubDate>
    </item>
    <item>
        ...
    </item>
</channel>
</rss>

```

RSS 是一种非常容易理解的文档协议, 将上述文档格式用中文重新标识其中的内容和作用就可以了解整个文档的数据结构了:

```

<rss version="2.0">
    <channel>
        <title>频道标题名称</title>
        <image>
            <title>图片标题</title>
            <link>图片链接地址</link>
            <url>图片地址</url>
        </image>
        <description>新闻频道的详细描述
        </description>
        <link>本频道的链接地址</link>
        <language>语言</language>
        <item>
            <title>新闻标题</title>

```





```

        <link>新闻的原文链接</link>
        <description>新闻摘要信息
        </description>
        <pubDate>发布日期</pubDate>
    </item>
    <item>
        ...
    </item>
</channel>
</rss>

```

是不是比较容易理解?

当然 RSS 的所有元素定义远远不止这么一点, 为了表述方便, 此处不对其全部的元素定义一一赘述了, 有兴趣的读者可以到 W3CSchool 的 RSS 参考手册 ([http://www.w3school.com.cn/rss/rss\\_reference.asp](http://www.w3school.com.cn/rss/rss_reference.asp)) 上阅读更多内容。

读懂我们需要爬取的目标数据格式和数据内容之后就可以动手设计爬虫项目了。

首先, 用 `urlopen` 打开目标 URL (<http://www.chinanews.com/rss/scroll-news.xml>) 并加载到 `BeautifulSoup` 对象内:

```

response = urlopen('http://www.chinanews.com/rss/scroll-news.xml')
rss = BeautifulSoup(response.read(), "html.parser")

```

然后, 调用 `find_all` 方法将所有 `<item>` 标记内的数据提取出来保存到一个字典中:

```

items = [] # 结果数据

for item in rss.find_all('item'):

    feed_item = {
        'title': item.title.text,      # 新闻标题
        'link': item.link.text,        # 原文链接
        'desc': item.description.text, # 新闻摘要
        'pub_date': item.pubdate.text  # 发表时间
    }

    items.append(feed_item)

```



最后，将 `items` 对象序列化为一个 JSON 字符串保存至文件内：

```
with open('result.json', 'wt') as file:
    file.write(json.dumps(items))
```

完整代码如下所示。

```
# -*- encoding: utf8 -*-
from urllib import urlopen
from bs4 import BeautifulSoup
import json

response = urlopen('http://www.chinanews.com/rss/scroll-news.xml')
rss = BeautifulSoup(response.read(), "html.parser")

items = [] # 结果数据

for item in rss.find_all('item'):
    feed_item = {
        'title': item.title.text,
        'link': item.link.text,
        'desc': item.description.text,
        'pub_date': item.pubdate.text
    }

    items.append(feed_item) # 将结果保存到 JSON 文件内

with open('result.json', 'wt') as file:
    file.write(json.dumps(items))
```

将以上的代码保存到 `crawl_chinanews.py`，然后在命令行运行以下代码：

```
$ python crawl_chinanews.py
```

此时可能会觉得这个程序只不过仅仅获取了新闻供稿上一个链接下的内容而已，是否可以将所有的供稿内容一次性都爬取下来呢？接下来，我们就将这个例子进行一次扩展，这时就需要对这个爬虫进行一次重新命题了：

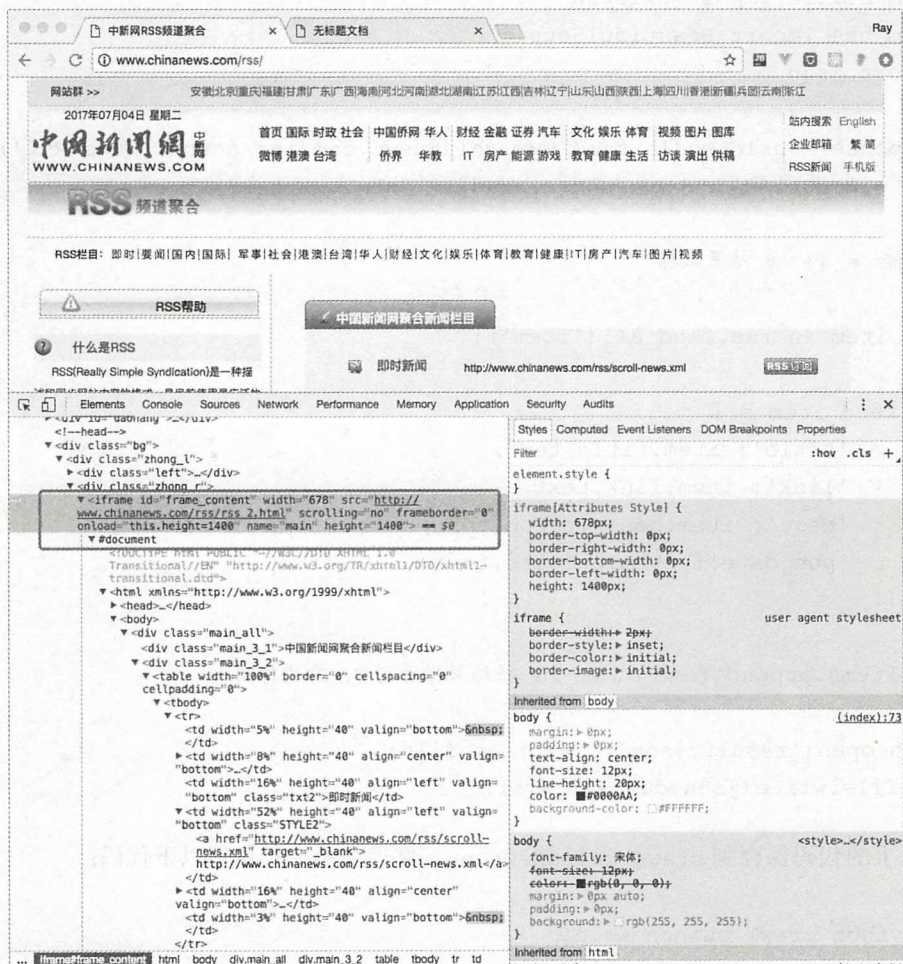




从 <http://www.chinanews.com/rss> 上获取所有可爬取的新闻供稿链接,然后将所有链接的内容读出并保存到本地。

我们称这种在一个已知的 URL 上获取真正需要爬取的数据目标的方式为间接式爬取,而上例中的爬取方式则是直接式爬取。

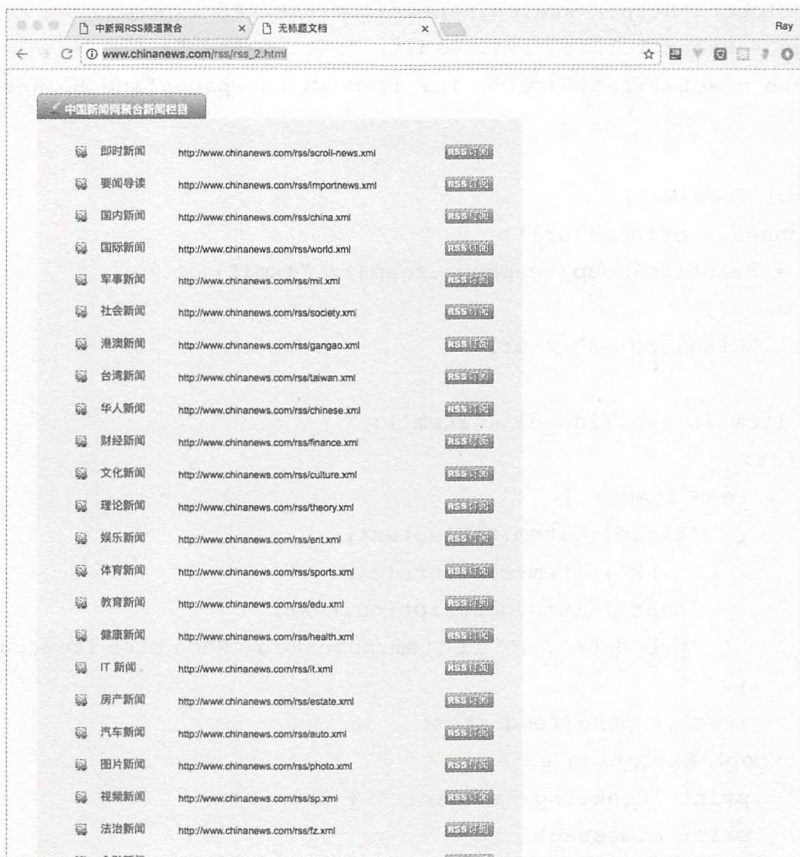
接下来我们将采用间接式爬取方式来重构上例: 打开 <http://www.chinanews.com/rss> 的代码结构查看供稿链接的规则, 如下图所示。



不难发现存放 RSS 链接的元素是被放在一个 `<iframe>` 元素内的, 也就是说, 我们看到的网页内容是由两个页面所构成的, 如果直接爬取 <http://www.chinanews.com/rss>, 则会毫无所



获，因为链接根本不在此页上。复制<iframe>元素上 src 属性内的 URL 并在新的窗口打开，同样用开发人员工具查看页面元素，如下图所示。



这才是真正的新闻供稿页面！

我们需要从这个页面中分析出新闻供稿链接的地址的查询规则。如果一次性将所有的 a 元素取出，则不免会出现重复性数据，如果要准确取出，则需要精准的 CSS 选择器进行定位。我喜欢用最简单的办法写代码，大道至简。所以我的方法是用 Python 自身的语言特性来排除这种数据重复，其实就是所谓的“去重”。Python 有一个特殊的数组类型叫 set，这个类型的特点就是会自动去除重复的数据。也就是说，可以一次性将所有的链接放到一个 set 变量中，而转换为 set 的变量是会自动排除出重复的数据的，那么这个变量内的链接其实就是我们真正需要的内容了。

```
from urllib import urlopen
from bs4 import BeautifulSoup
```





```
import json

res = urlopen('http://www.chinanews.com/rss/rss_2.html')
rss_page = BeautifulSoup(res.read(), "html.parser")
rss_links = set([item['href'] for item in rss_page.find_all('a')])

def crawl_feed(url):
    response = urlopen(url)
    rss = BeautifulSoup(response.read(), "lxml")
    items = []
    print "Crawling %s" % url

    for item in rss.find_all('item'):
        try:
            feed_item = {
                'title': item.title.text,
                'link': item.contents[2],
                'desc': item.description.text,
                'pub_date': u'' if item.pubdate is None else item.pubdate.text
            }
            items.append(feed_item)
        except Exception as e:
            print 'Crawling %s error.' % url
            print e.message

    return items

feed_items = []

for link in rss_links:
    feed_items += crawl_feed(link)

with open('result.json', 'a') as file:
    file.write(json.dumps(feed_items))

print 'Total crawl %s items' % len(feed_items)
```



以上代码将前一个示例代码重构为一个 `crawl_feed` 的函数，将一个新闻供稿看作动作单元。然后对经过重处理的目标链接 `rss_links` 进行循环处理，在循环体内调用 `crawl_feed` 函数对真正的新闻数据进行爬取。这就是所谓的**间接式爬网**的实现方式。

## 1.5 小结

本章介绍了虫术的一些基本背景知识，用中新网的一个简单爬虫解释组成爬虫系统的最小化结构和基本开发思路，主要分为以下三步：

(1) 产生爬取目标。

- 直接式——具有明确且具体的 URL 目标，全面无差别地爬取；
- 间接式——以一个或多个已知的 URL 作为入口，然后从中获取真正具体的爬取目标。

(2) 通过循环完成爬取。

- 最原始的就是单线程的循环；
- 可以通过协程处理实现并发式循环。

(3) 分析、提取爬取内容并进行存储。

以上就是爬虫系统的基本设计开发的步骤与思路，其他的爬虫框架也是在这个基础之上进行的功能抽象、扩展延伸，以及性能和稳定性的加固。





# 2 chapter

## 第 2 章

# Scrapy 基础知识

Scrapy 算得上是 Python 世界上最常用的爬虫框架了，同时它也是我掌握的几种流行语言中最好的爬虫框架，没有之一！我认为它也是最难学习的框架，同样没有之一。很多初学 Scrapy 的同事经常向我抱怨完全不清楚 Scrapy 该怎样入手，即使看的是中文的文档，也感到很难理解。我当初接触 Scrapy 时也有这样的感觉。之所以感到 Scrapy 难学，究其原因，是其官方文档实在太过凌乱，又缺少实用的代码例子，让人看得云里雾里，不知其所以然。

虽然其文档不良，但却没有遮挡住它的光辉，它依然是 Python 世界中目前最好用的爬虫框架。其架构的思路、蜘蛛执行的效能，还有可扩展的能力都非常出众，再配以 Python 语言的简洁轻巧，使得爬虫的开发事半功倍。

因为 Scrapy 以一种并发式的异步结构来组织各个模块的运行，所以我会先以简单的示例来讲解 Scrapy 的组成与运行方式，再对 Scrapy 架构进行综合、全面的解释，然后细讲 Scrapy 中每个模块的概念及其作用。

我并不是一个爱屋及乌之人，在实际运用 Scrapy 的过程中我发现它有很多的不足，甚至在官方文档中，某些推荐用法其实是一些“大坑”。因此，本书只讲述 Scrapy 中最有用同时也是最重要的模块内容，大多的 Scrapy 提供的内置类都没有介绍，在阅读到具体的内容时会说明不建议使用的原因。

### Scrapy 的优点

以下是我选择 Scrapy 作为虫术的核心框架而不选用其他框架的理由：

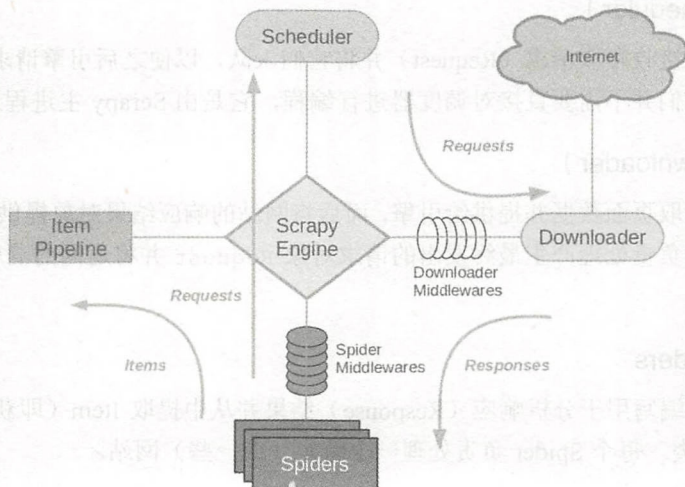
- 提供了内置的 HTTP 缓存，以加速本地开发。



- 提供了自动节流调节机制，而且具有遵守 robots.txt 的设置的能力。
- 可以定义爬行深度的限制，以避免爬虫进入死循环链接。
- 会自动保留会话。
- 执行自动 HTTP 基本认证。不需要明确保存状态。
- 可以自动填写登录表单。
- Scrapy 有一个内置的中间件，可以自动设置请求中的引用（referrer）头。
- 支持通过 3xx 响应重定向，也可以通过 HTML 元刷新。
- 避免被网站使用的<noscript> meta 重定向困住，以检测没有 JS 支持的页面。
- 默认使用 CSS 选择器或 XPath 编写解析器。
- 可以通过 Splash 或任何其他技术（如 Selenium）呈现 JavaScript 页面。
- 拥有强大的社区支持和丰富的插件和扩展来扩展其功能。
- 提供了通用的蜘蛛来抓取常见的格式：站点地图、CSV 和 XML。
- 内置支持以多种格式（JSON、CSV、XML、JSON-lines）导出收集的数据并将其存储在多个后端（FTP、S3、本地文件系统）中。

## 2.1 Scrapy架构

有了第1章中的爬虫示例作为基础，学习 Scrapy 就非常容易了，前文中的爬虫示例只是为了说明一个爬虫程序应该具有的最小模块功能。而 Scrapy 则是基于这种爬虫理论下的完整爬虫开发框架。首先我们可以通过 Scrapy 的体系架构从抽象层级的角度来了解它提供了哪些具体的模块，这些模块又是如何相互协同工作的。以下截取的是 Scrapy 官方文档中的系统架构图。





Scrapy 中的数据流由执行引擎控制, 其过程如下:

- (1) 引擎打开一个网站 (open a domain), 找到处理该网站的 Spider 并向该 Spider 请求第一个要爬取的 URL (s)。
- (2) 引擎从 Spider 中获取第一个要爬取的 URL 并在调度器 (Scheduler) 中以 Request 调度。
- (3) 引擎向调度器请求下一个要爬取的 URL。
- (4) 调度器返回下一个要爬取的 URL 给引擎, 引擎将 URL 通过下载中间件 (请求 (request) 方向) 转发给下载器 (Downloader)。
- (5) 一旦页面下载完毕, 下载器生成一个该页面的 Response, 并将其通过下载中间件 (返回 (Response) 方向) 发送给引擎。
- (6) 引擎从下载器中接收 Response 并通过 Spider 中间件 (输入方向) 发送给 Spider 处理。
- (7) Spider 处理 Response 并返回爬取到的 Item 及 (跟进的) 新的 Request 给引擎。
- (8) 引擎将 (Spider 返回的) 爬取到的 Item 给 Item Pipeline, 将 (Spider 返回的) Request 给调度器。
- (9) (从第 2 步) 重复直到调度器中没有更多的 Request, 引擎关闭对该网站的执行进程。

## Scrapy Engine

引擎负责控制数据流在系统中所有组件中流动, 并在相应动作发生时触发事件。它也是程序的入口, 可以通过 scrapy 指令方式在命令行启动, 或者通编程方式实例化后调用 start 方法启动。

### 调度器 (Scheduler)

调度器从引擎接收爬取请求 (Request) 并将它们入队, 以便之后引擎请求它们时提供给引擎。一般来说, 我们并不需要直接对调度器进行编程, 它是由 Scrapy 主进程进行自动控制的。

### 下载器 (Downloader)

下载器负责获取页面数据并提供给引擎, 而后将网站的响应结果对象提供给蜘蛛 (Spider)。具体点说, 下载器负责处理产生最终发出的请求对象 Request 并将返回的响应生成 Response 对象传递给蜘蛛。

### 蜘蛛——Spiders

Spider 是用户编写用于分析响应 (Response) 结果并从中提取 Item (即获取的 Item) 或额外跟进的 URL 的类。每个 Spider 负责处理一个特定 (或一些) 网站。

### 数据管道——Item Pipeline

Item Pipeline 负责处理被 Spider 提取出来的 Item。典型的处理有清理、验证及持久化（例如，存取到数据库中）。

### 下载器中间件（Downloader middlewares）

下载器中间件是在引擎及下载器之间的特定钩子（specific hook），处理 Downloader 传递给引擎的 Response。其提供了一个简便的机制，通过插入自定义代码来扩展 Scrapy 的功能。

### Spider中间件（Spider middlewares）

Spider 中间件是在引擎及 Spider 之间的特定钩子（specific hook），处理 Spider 的输入（Response）和输出（Items 及 Requests）。其提供了一个简便的机制，通过插入自定义代码来扩展 Scrapy 的功能。

从 Scrapy 的系统架构可见，它将整个爬网过程进行了非常具体的细分，并接管了绝大多数复杂的工作，例如，产生请求和响应对象、控制爬虫的并发等。

## 2.2 Scrapy快速入手

从总体架构上了解了 Scrapy 之后，相信读者已经建立了对 Scrapy 基本的整体认识。接下来还是从实践入手，通过实践可以快速地了解 Scrapy 的使用方法。

### 安装Scrapy

本节的安装步骤假定读者已经安装好下列程序。

- Python 2.7。
- Python Package: 现在 pip 依赖 setuptools，如果未安装，则会自动安装 setuptools。
- lxml: 大多数 Linux 发行版自带了 lxml。如果缺失，则查看(<http://lxml.de/installation.html>)。
- OpenSSL: 除 Windows 外的系统都已经提供。

可以使用 pip 来安装 Scrapy（推荐使用 pip 来安装 Python package）。

使用 pip 安装：

```
$ pip install Scrapy
```

### Scrapy tool

按照上面的安装方法会把 Scrapy 安装到系统的全局目录中。之所以将 Scrapy 安装到系统的全局目录中而不采用 VirtualEnv，是因为 Scrapy 除了提供一系列的 Python 库，也是一个命令



行工具，只有安装到全局目录，我们在系统中的所有文件夹下才能正常地调用它。

可以在命令行中键入：

```
$ scrapy
```

会看到以下的提示内容：

```
Scrapy 1.0.3 - project: scrapy_quickstart

Usage:
  scrapy <command> [options] [args]

Available commands:
  bench          Run quick benchmark test
  check          Check spider contracts
  commands
  crawl          Run a spider
  edit           Edit spider
  fetch          Fetch a URL using the Scrapy downloader
  genspider      Generate new spider using pre-defined templates
  list           List available spiders
  parse          Parse URL (using its spider) and print the results
  runspider      Run a self-contained spider (without creating a project)
  settings       Get settings values
  shell          Interactive scraping console
  startproject   Create new project
  version        Print Scrapy version
  view           Open URL in browser, as seen by Scrapy
```

Scrapy 提供的这一系列丰富的命令工具的内容很多，在下文中会一一用到。详细的使用方法与示例请见本章的“附：Scrapy 工具命令参考”。

在创建 Scrapy 的爬网项目时并不需要手动为 Scrapy 项目准备基本的文件与文件夹，只需要运行 startproject 指令参数就可以直接初始化 Scrapy 项目，具体指令如下：

```
$ scrapy startproject chinanews_crawler
```

对于初学者而言，推荐使用 Scrapy 提供的命令行工具来初始化项目，这样能从官方所推荐

的结构中快速了解最小运行框架中实质上只需要哪些模块。

### ► 文件目录结构

以下是执行 `scrapy startproject <项目名称>`（此处的项目名称为 `chinanews_crawler`）指令后，由 Scrapy 所创建的项目目录及文件结构：

```
.
├── chinanews_crawler
│   ├── __init__.py    ## 包定义
│   ├── items.py       ## 模型定义
│   ├── pipelines.py   ## 管道定义
│   ├── settings.py    ## 配置文件
│   └── spiders        ## 蜘蛛文件夹
│       └── __init__.py ## 默认的蜘蛛代码文件
└── scrapy.cfg         ## Scrapy 的运行配置文件
```

Scrapy 创建的文件结构都非常明确，每个文件的具体内容在此暂且不表，下文自有交代。现在我们就将第1章中的示例用 Scrapy 重新改写，看看 Scrapy 是如何实现之前示例中的爬网逻辑的，从两者的区别中就能快速理解 Scrapy 的用法。

需要说明一下的是 `scrapy.cfg` 和 `settings.py` 两个配置文件。`scrapy.cfg` 存放的目录被认为是项目的根目录。该文件中包含 Python 模块名的字段定义了项目的设置，而 `settings.py` 则是通过 Python 代码以编程方式控制的配置文件。打开 `scrapy.cfg` 就能知道二者的关系：

```
[settings]
default = chinanews_crawler.settings
```

`scrapy.cfg` 属于发布后的运行配置，它用于指向具体爬网时采用的 Python 配置代码，`chinanews_crawler` 文件夹内带有一个 `__init__.py` 文件，声明了这个文件夹是一个 Python 包，引用该包下的模板采用 `chinanews_crawler.文件名` 的方式。因此在 `scrapy.cfg` 内，会采用 `chinanews_crawler.settings` 的引用方式指向 `settings.py` 文件。

### ► 创建蜘蛛

蜘蛛 (Spider) 是 Scrapy 中很重要也是很常用的一个功能，当我们创建一个 Scrapy 项目时，一般都会先编写蜘蛛的逻辑。蜘蛛的作用是分析由 Scrapy 引擎返回的爬网内容，并决定是否继续生成新的爬网请求。关于蜘蛛的概念和详细解释会在“蜘蛛 Spider”一节详细讲述。



Scrapy 提供了一个快速生成蜘蛛命令的工具 `genspider`，蜘蛛的命名一般会采用与爬取的网站相同的名称以便于辨识，例如：

```
$ scrapy genspider chinanews chinanews.com
```

**注意：**`genspider` 是一个项目命令，因此要先进入上文采用 `startproject` 创建的项目目录内才能正常运行。

第一个参数是蜘蛛的名称，第二个参数是指定 Scrapy 爬网的起始位置。

当 `genspider` 命令成功执行后，会在 `chinanews_crawler` 目录下多出一个 `spiders` 的 Python 包，其中有一个名为 `chinanews.py` 的 Python 代码文件。该文件的具体内容如下所示。

```
# -*- coding: utf-8 -*-
import scrapy

class ChinanewsSpider(scrapy.Spider):
    name = "chinanews"
    allowed_domains = ["chinanews.com"]
    start_urls = (
        'http://www.chinanews.com/rss/',
    )

    def parse(self, response):
        pass
```

这个蜘蛛除了会向 `http://www.chinanews.com/rss/` 发送请求，其他什么也不会干，因为 `parse` 函数内什么也没有。那应该怎么将之前的示例代码用 Scrapy 的蜘蛛来改写呢？此处恕我先卖个关子，在接下来的“蜘蛛”一节中会详细地讲述。

### ➤ 启动爬网

Scrapy 的启动非常简单，只要打开命令行并进入当前的项目目录中运行以下指令即可：

```
$ scrapy crawl chinanews
```

以上指令是告知 Scrapy 使用名为 `chinanews` 的蜘蛛进行爬网。

这个命令将默认运行项目配置指定的所有蜘蛛的爬取任务，具体效果如下图所示。

```

(venv) RayOSX:chinanews_crawler Ray$ scrapy crawl chinanews
2018-03-21 22:00:10 [scrapy.utils.log] INFO: Scrapy 1.4.0 started (bot: chinanews_crawler)
2018-03-21 22:00:10 [scrapy.utils.log] INFO: Overridden settings: {'NEWSPIDER_MODULE': 'chinanews_crawler.spiders', 'FEED_FORMAT': 'json', 'SPIDER_MODULES': ['chinanews_crawler.spiders.chinanews'], 'FEED_URI': 'result.json', 'BOT_NAME': 'chinanews_crawler'}
2018-03-21 22:00:10 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.feedexport.FeedExporter',
'scrapy.extensions.memusage.MemoryUsage',
'scrapy.extensions.logstats.LogStats',
'scrapy.extensions.telnet.TelnetConsole',
'scrapy.extensions.corestats.CoreStats']
2018-03-21 22:00:10 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.httppath.HttpAuthMiddleware',
'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
'scrapy.downloadermiddlewares.retry.RetryMiddleware',
'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
'scrapy.downloadermiddlewares.stats.DownloaderStats']
2018-03-21 22:00:10 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
'scrapy.spidermiddlewares.referrer.RefererMiddleware',
'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
'scrapy.spidermiddlewares.depth.DepthMiddleware']
2018-03-21 22:00:10 [scrapy.middleware] INFO: Enabled item pipelines:
['chinanews_crawler.pipelines.BlockGamePipeline',
'chinanews_crawler.pipelines.CleanHTMLPipeline',
'chinanews_crawler.pipelines.JsonFeedPipeline']
2018-03-21 22:00:10 [scrapy.core.engine] INFO: Spider opened
2018-03-21 22:00:10 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min)
2018-03-21 22:00:10 [scrapy.extensions.telnet] DEBUG: Telnet console listening on 127.0.0.1:6023
2018-03-21 22:00:11 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://www.chinanews.com/rss/rss_2.html> (referrer: None)
2018-03-21 22:00:11 [scrapy.core.engine] INFO: Closing spider (finished)
2018-03-21 22:00:11 [scrapy.statscollectors] INFO: Dumping Scrapy stats:
{'downloader/request_bytes': 229,
'downloader/request_count': 1,
'downloader/request_method_count/GET': 1,
'downloader/response_bytes': 2210,
'downloader/response_count': 1,
'downloader/response_status_count/200': 1,
'finish_reason': 'finished',
'finish_time': datetime.datetime(2018, 3, 21, 14, 0, 11, 875892),
'log_count/DEBUG': 2,
'log_count/INFO': 7,
'memusage/max': 48930816,
'memusage/startup': 48930816,
'response_received_count': 1,
'scheduler/dequeued': 1,
'scheduler/dequeued/memory': 1,
'scheduler/enqueued': 1,
'scheduler/enqueued/memory': 1,
'start_time': datetime.datetime(2018, 3, 21, 14, 0, 10, 437304)}
2018-03-21 22:00:11 [scrapy.core.engine] INFO: Spider closed (finished)
(venv) RayOSX:chinanews_crawler Ray$

```

这个运行图内包含很多信息，在以后的运用中它将是一个非常重要的信息观察来源。

## 2.3 数据模型Item

在讲述蜘蛛之前还得先谈谈 Item，因为所有的蜘蛛都需要使用它，以下是摘录的 Scrapy 官网对 Item 的诠释：

爬取的主要目标就是从非结构性的数据源中提取结构性数据，如网页。Scrapy 提供 Item 类来满足这样的需求。Item 对象是一种简单的容器，保存了爬取到的数据。其提供了类似于词典的 API，以及用于声明可用字段的简单语法。



我想从另一个角度来解释 Item 的概念，首先 Item 是一种数据容器，是作为蜘蛛与管道之间的数据载体，蜘蛛对收集的数据结构进行分析后提取出具体的数据结构并生成对应的 Item 实例，然后由 Scrapy 引擎传递给对应的管理进行后处理。

这样解释是不是很烦琐？如果读者有这个感觉，则证明读者已经完全了解了第 1 章中 FeedItem 的作用。

### 声明 Item

由于 Item 是一个接口类（虽然 Python 没有接口的概念，但可以这样来理解），因此我们得像定义数据库的数据模型一样来定义 Item 类。Item 使用简单的 class 定义语法和 Field 对象来进行声明。

### Item 字段 (Item Fields)

Field 对象指明了每个字段的元数据 (metadata)。例如，下面例子的 last\_updated 中指明了该字段的序列化函数。

可以为每个字段指明任何类型的元数据。Field 对象对接受的值没有任何限制。也正是因为这个原因，文档也无法提供所有可用的元数据的键 (key) 参考列表。Field 对象中保存的每个键可以由多个组件使用，并且只有这些组件知道这个键的存在。读者可以根据自己的需求，定义使用其他的 Field 键。设置 Field 对象的主要目的就是在在一个地方定义好所有的元数据。一般来说，那些依赖某个字段的组件肯定使用了特定的键 (key)。我们必须查看组件相关的文档，查看其用了哪些元数据键 (metadata key)。

```
import scrapy

class Product(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field()
    stock = scrapy.Field()
    last_updated = scrapy.Field(serializer=str)
```

需要注意的是，用来声明 Item 的 Field 对象并没有被赋值为 class 的属性。不过可以通过 Item.fields 属性进行访问。

以上就是如何声明 Item 的内容。

### 实践示例

接下来将第 1 章中的 FeedItem 改写成 Scrapy 的 Item:

```
from scrapy.item import Item, Field
```

```
class FeedItem(Item):
    title = Field()
    link = Field()
    desc = Field()
    pub_date = Field()
```

只要将 `FeedItem` 继承至 `scrapy.item.Item` 类，每个属性的默认值都构造一个 `Field()` 实例就行了。

如果深入 `Item` 的代码(在 PyCharm 中按住 `Ctrl` 键，然后单击 `Item` 就可以打开 `Item` 的源码，或者在 `lib/python/lib/site-packages/scrapy/item.py` 中找到它的身影)，就会发现 `Field` 类只是简单地继承至一个字典类 (`dict`)。同样，`Item` 是继承自 `DictItem` 这个与字典类用法非常相似的类，也就是说，当 `Item` 被实例化后直接将其作为字典来使用就行了。

#### ➤ 设置字段的值

```
news_item = FeedItem()
news_item['title'] = feedItem.title
news_item['link'] = feedItem.find('link')
```

#### ➤ 获取字段的值

```
>>> news_item['title']
云南森警开展普法宣传 一天收缴 29 支枪
>>> news_item['link']
http://www.chinanews.com/sh/2017/07-04/8269059.shtml
```

#### ➤ 枚举字段的键-值

可以使用 `dict` 来获取所有的值：

```
>>> news_item.keys()
['title', 'link', 'desc', 'pub_date']

>>> news_item.items()
[('title', u'云南森警开展普法宣传 一天收缴 29 支枪'), ('link', 'http://www.chinanews.com/sh/2017/07-04/8269059.shtml'), ('desc', u'...'), ('pub_date', '2017-07-04 23:03:51')]
```



## ➤ 其他任务

复制 Item:

```
>>> news_item2 = FeedItem(news_item)
>>> print news_item2
FeedItem(title=u'云南森警开展普法宣传 一天收缴 29 支枪', link='http://www.chinanews.com/sh/2017/07-04/8269059.shtml', desc='...', pub_date='2017-07-04 23:03:51')

>>> news_item3 = news_item2.copy()
>>> print news_item3
FeedItem(title=u'云南森警开展普法宣传 一天收缴 29 支枪', link='http://www.chinanews.com/sh/2017/07-04/8269059.shtml', desc='...', pub_date='2017-07-04 23:03:51')
```

根据 Item 创建字典 (dict):

```
>>> dict(news_item)
```

这种做法在将 Item 转化为 JSON 对象时非常有用，下文中将会提及。

根据字典 (dict) 创建 item:

```
>>> FeedItem({'title':u'云南森警开展普法宣传 一天收缴 29 支枪',
              'pub_date':'2017-07-04 23:03:51'})
FeedItem(title=u'云南森警开展普法宣传 一天收缴 29 支枪', pub_date='2017-07-04 23:03:51')
```

## 扩展 (继承) Item

可以通过继承原始的 Item 来扩展 (继承)Item (添加更多的字段或者修改某些字段的元数据)。

例如:

```
class RssFeedItem(FeedItem):
    author = scrapy.Field(serializer=str)
    permalink = scrapy.Field()
```

也可以通过使用原字段的元数据来添加新的值，或者修改原来的值来扩展字段的元数据:



```
class SpecificFeedItem(FeedItem):  
    name = scrapy.Field(FeedItem.fields['title'], serializer=my_serializer)
```

上述代码在保留所有原来的元数据值的情况下添加（或者覆盖）了 `title` 字段的 `serializer`。

看了这么多，读者是否觉得 Scrapy 提供的 Item 定义非常麻烦呢？为何不直接像在第 1 章那样定义一个类来作为数据容器，非要搞出这么多的用法呢？原因在于这些 Item 数据的产生都要被 Scrapy 跟踪和序列化。虽然写起来有点麻烦，但编程有时就得按照框架所制定的规则来做。

## 2.4 蜘蛛——Spiders

Spider 类定义了如何爬取某个（或某些）网站。包括爬取的动作（例如：是否跟进链接），以及如何从网页的内容中提取结构化数据（爬取 Item）。换句话说，Spider 就是定义爬取的动作及分析某个网页（或者是有些网页）的地方。

对于 Spider 来说，爬取的循环一般是这样的：

(1) 以初始的 URL 初始化 request，并设置回调函数。当该 request 下载完毕并返回时生成 response，并作为参数传给该回调函数。Spider 中初始的 request 是通过调用 `start_requests()` 来获取的。`start_requests()` 函数会读取 `start_urls` 中的 URL，并以 `parse` 为回调函数生成 request。

(2) 在回调函数中分析返回的（网页）内容，返回 Item 或者一个包括二者的可迭代容器。返回的 Request 对象之后会经过 Scrapy 处理，下载相应内容，并调用设置的 `callback` 函数（函数可相同）。

(3) 在回调函数中，可以使用选择器（Selectors）（也可以使用 BeautifulSoup、lxml 或者想用的任何解析器）来分析网页内容，并根据分析的数据生成 Item。

(4) 由 Spider 返回的 Item 将被存到数据库（由某些 Item Pipeline 处理）或者使用 Feed exports 保存到文件中。

虽然该循环对任何类型的 Spider 都（多少）适用，但 Scrapy 仍然为了不同的需求提供了多种默认的 Spider。之后将讨论这些 Spider。

我们现在就来将第 1 章示例中的代码移植到蜘蛛中，首先打开之前用 Scrapy 工具创建的 `chinanews.py` 蜘蛛代码文件：

```
# -*- coding: utf-8 -*-  
from scrapy import Spider
```





```
class ChinanewsSpider(Spider):
    name = "chinanews"
    allowed_domains = ["chinanews.com"]
    start_urls = (
        'http://www.chinanews.com/rss/scroll-news.xml',
    )

    def parse(self, response):
        pass
```

蜘蛛是 Scrapy 项目中第二个需要自定义的类, 对照以上的代码, 先介绍一下蜘蛛的基本结构。首先, 蜘蛛要从 Spider 基类中继承, 当然也可以从 Spider 的其他子类继承过来。Scrapy 工具所创建的蜘蛛类有三个属性:

- name——蜘蛛名称, 可用于被 Scrapy 命令工具识别。例如, `$ scrapy crawl chinanews`, 当一个项目中同时存在多个蜘蛛时, 这个名称标识尤为重要。
- allowed\_domains——只爬取该域内的内容, 自动过滤链接到其他域的内容。
- start\_urls——当蜘蛛被启动并产生第一批请求时的 URL 数组。

其次, 等待我们实现的就是 parse 函数。当第一批请求被下载器发向目标地址并得到响应结果时将被调用, 因此这个函数就是用于分析响应结果的。将中新网爬虫的分析代码加入 parse 函数后, 代码如下所示。

```
def parse(self, response):
    rss = BeautifulSoup(response.body, "html.parser")
    for item in rss.find_all('item'):
        feed_item = FeedItem()
        feed_item['title'] = item.title
        feed_item['link'] = item.link
        feed_item['desc'] = item.description
        feed_item['pub_date'] = item.pubdate
        yield feed_item
```

初学 Python 的读者可能不太理解 yield 关键字的作用, yield 有点类似于 return, 都是用于返回结果的。与 return 不同的是, yield 返回的是一个迭代器而不是具体值, 且 yield 不会像 return 直接将当前执行代码中断并返回, 而是将当前可被返回的对象生成一个迭代器, 然后继续执行下一行的代码。



`parse` 函数内的 `response` 是由下载器生成并传入的，我们将其原生的文本内容传入 `BeautifulSoup` 构造函数并生成 `BeautifulSoup` 实例，而 `FeedItem` 则以上一节介绍的方式对其属性进行写入。最后返回 `FeedItem` 结果对象的迭代器。

此时第一个示例中的“直接式”爬网的逻辑就完成了，而要将数据保存到 JSON，则 Scrapy 可以用一种无代码的方法实现，只要按以下方式启动蜘蛛爬网即可：

```
$ scrapy crawl -o result.json
```

`-o` 参数是指“输出到文件”。

接下来尝试实现“间接式”爬网用的蜘蛛，在此之前先回顾一下“间接式”爬网的实现逻辑：

(1) 从 `http://www.chinanews.com/rss/rss_2.html` 中读取 RSS 链接。

(2) 逐一向所有的 RSS 链接发出请求并分析具体响应内容提取出 `FeedItem`。

按照这个逻辑，`start_urls` 中存放的并不是包含的目标数据，需要在响应的内容中提取并进行二次请求，那么这个蜘蛛就需要有两个 `parse` 函数实现这种“间接式”爬取数据的逻辑了，一个负责爬取目标链接作为索引，另一个则负责爬取数据项：

```
from scrapy.spiders import Spider
from scrapy.http import Request
from ..items import NewsFeedItem
from bs4 import BeautifulSoup

class ChinaNewsSpider(Spider):
    name = "chinanews"
    start_urls = (
        'http://www.chinanews.com/rss/rss_2.html',
    )

    def parse(self, response):
        rss_page = BeautifulSoup(response.body, "html.parser")
        rss_links = set([item['href'] for item in rss_page.find_all('a')])
        for link in rss_links:
            yield Request(url=link, callback=self.parse_feed)
```





```
def parse_feed(self, response):  
    rss = BeautifulSoup(response.body, 'lxml')  
    for item in rss.find_all('item'):  
        feed_item = FeedItem()  
        feed_item['title'] = item.title.text  
        feed_item['link'] = item.link.text  
        feed_item['desc'] = item.description.text  
        feed_item['pub_date'] = item.pubdate.text  
  
        yield feed_item
```

为了避免混淆两个 `parse` 函数，在此先稍做一点说明。`parse` 是由 `Spider` 基类中定义的一个空函数，打开 `Spider` 的代码就会发现这个函数体内只有一个 `pass` 关键字，其他什么事都不做。我们可以将 `Spider` 理解为一个抽象类，而 `parse` 则是所有子类中必须实现的“抽象方法”（Python 中并没有抽象类与抽象方法这么一说，但可以借此概念触类旁通）。因此，每个蜘蛛都必须重写这个 `parse` 函数，`Scrapy` 的调度器会找到它并自动调用。

上述代码中的 `parse_feed` 函数则是一个自定义函数，在 `parse` 函数中不再返回 `FeedItem` 的迭代器，而是返回了一个 `request` 的迭代器，在 `request` 的构造函数中将 `parse_feed` 函数引用作为构造参数传入。当 `request` 对象被下载器发送到目标 URL 并返回之后，就会自动执行 `parse_feed` 并传入相应的 `response`，这样一来就形成了一种二次循环。在虫术上来说就是“深度”，在本示例中只进入了一层深度。用直接的方式来理解就是一个层深度就需要有一个 `parse` 函数产生下一层次请求对象，当不再产生新的请求对象就意味着这种深度循环的终结。

最后，当 `parse` 方法返回的是一个 `Item` 的枚举时，标志着这个蜘蛛已经完成它需要完处理的事情了，`Scrapy` workflow 将移交到一个处理步骤中去，也就是下一节要讲述的“管道”。

## 2.5 管道——Item Pipeline

当 `Item` 在 `Spider` 中被收集之后，它将被传递到管道中（`Item Pipeline`），一些组件会按照一定的顺序对 `Item` 进行处理。

每个管道组件是实现了简单方法的 Python 类。它们接收 `Item` 并通过 `Item` 执行一些行为，同时也决定此 `Item` 是否继续通过管道，或是被丢弃而不再进行处理。

以下是管道的一些典型应用：

- 清理 HTML 数据；



- 验证爬取的数据（检查 Item 包含某些字段）；
- 去重（并丢弃）；
- 将爬取结果保存到数据库中。

除了以上的几种应用，其实管道可以处理任意与 Item 相关（丢弃与继续）的事务。之所以叫管道，是因为它只负责管输入和输出，并且输入的对象都是 Item。

编写自定义的管道非常简单，每个管道都是一个具有 `process_item` 方法的 Python 类，具体如下所示。

```
class MyPipeline(object):  
    def process_item(self, item, spider):  
        return item
```

每个管道都需要调用 `process_item` 方法，而且这个方法必须返回一个 Item 对象或是抛出一个 `DropItem` 异常。以下是 `process_item` 方法的参数说明：

- `item`——当前处理的 Item 对象；
- `spider`——产生当前 Item 对象的蜘蛛实例。

**注意：**管道每次只处理一个 Item 对象。

接下来将采用管道的方式来实现本章示例中最后的处理部分，并且为了更好地帮助读者理解管道的作用，我会按管道的应用特性分别实现几种不同的管道，来完成新闻供稿的去重、过滤、加工和存储处理。

管道按照应用特性来分类大致可以分为过滤性管道、加工性管道和存储性管道三种。

### 过滤性管道

在爬网的过程中经常会爬到一些对我们没有用或者是重复的数据，我们可以在存储这些 Item 数据之前加入一个管道来判断这些数据的可用性，保留有用的、丢弃无用的。只要在 `process_item` 处理方法中发起 `scrapy.exceptions.DropItem` 的异常就能完成丢弃的动作，并不用担心在 `process_item` 中引发异常会导致整个爬网过程的中止，因为管道的每次处理是针对单个 Item 对象进行的。

例如，我们不需要所有与“游戏”相关的信息，则可以编写一个这样的过滤性管道：

```
from scrapy.exceptions import DropItem
```





```
class BlockGamePipeline:
    def process_item(self, item, spider):
        filter_key = "游戏"
        if filter_key in (item['title']).encode('utf-8'):
            raise DropItem()
        return item
```

这个管道检查每个新闻供稿信息的标题是否带有“游戏”的关键字，如果有则丢弃，否则就将 Item 返回给 Scrapy 引擎并交由下一管道进行处理。管道的这种处理方式还可以帮助我们完成复杂的去重工作。假如我们认定相同的新闻标题其内容就是重复的，则可以将新闻标题作为一个唯一性的指纹数据，用 Python 的唯一数集 set 对象存储并进行简单的数据去重，代码如下所示。

```
class DuplicatesPipeline:
    def __init__(self):
        self.fingerprints = set()

    def process_item(self, item, spider):
        if self.fingerprints in item['title']:
            self.fingerprints.add(item["title"])
            raise DropItem()

        return item
```

set 是 Python 特有的唯一性数据集，可以理解为只具有唯一值的数据列表。

### 加工性管道

当我们在批量爬取数据后，需要对每一个 Item 内的数据进行一些额外的计算或者扩展时，就需要采用这种加工性管道。例如，当我们爬取一些商品信息时，就经常需要对单个产品的价格进行累加，以节省以后进行大批量的累加运算的时间，那么我们就可以在 Item 上预先附加一个累加后的字段，在流过管道时任意进行单体的计算：

```
class ProductPricePipeline:

    def process_item(self, item, spider):
        item['total'] = float(item['price']) * float(item['count'])
        return item
```



回到上面供稿例子中，为了防止 FeedItem 的 desc 或者 title 带有 HTML 标记（因为这些标记本质上对于我们来说是没用的，我们需要的是文字内容而并不关心内容的展现），我们可以加入一个清理 title 字段和 desc 字段内 HTML 标记的管道：

```
from bs4 import BeautifulSoup

class CleanHTMLPipeline:

    def clear_html(text):
        html = BeautifulSoup(text)
        return html.get_text()

    def process_item(self, item, spider):
        item['title'] = clear_html(item['title'])
        item['desc'] = clear_html(item['desc'])

    return item
```

### 存储性管道

将爬取的数据进行持久化处理的管道称为存储性管道，用来将 Item 保存为文件或者存储到数据库中，包括将图像文件及媒体文件下载为本地文件等都归入此类。

我们的示例之前是将数据保存到 JSON 文件中，下面来实现一个 JSON 的存储管道：

```
import json

class JsonFeedPipeline:

    def __init__(self):
        self.json_file = open('feed.json', 'wt')
        self.json_file.write("[\n")

    def process_item(self, item, spider):
        line = json.dumps(dict(item)) + ",\n"
        self.json_file.write(line)
        return item

    def close_spider(self, spider):
```





```
self.json_file.write("\n")
self.json_file.close()
```

实际上如果要将所有的 Item 保存到一个 JSON 文件中,则无须手工实现 JSON 的存储,此处只是用来说明其作用而已。而且要输出为 JSON 格式的文件,有以下两种办法:

- (1) 全局性指定——本示例之初已配置了 Scrapy 的文件存储后端,也就是前文中使用的 FEED\_URI 和 FEED\_FORMAT 两个配置项,Scrapy 会在执行完爬取工作后自动存储 JSON 文件到 FEED\_URI 指定的位置。
- (2) 动态指定——在 Scrapy 命令中加入 -o 的输出参数就可以使用 Scrapy 的数据导出器将数据输出到指定文件上。

可能读者会产生一个疑问:“这些管道是如何与蜘蛛关联起来的呢? Scrapy 能知道哪个蜘蛛对应哪个管道输出吗?”答案是否定的,蜘蛛与管道的关系需要通过配置文件来指定,当编写完这个管道的代码后就需要到 settings.py 文件中进行这个指定的动作。

打开 settings.py 文件,找到以下的配置内容:

```
# Configure item pipelines
# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html
ITEM_PIPELINES = {
#     'chinanews_crawler.pipelines.SomePipeline': 300,
# }
```

ITEM\_PIPELINES 内要以全路径引用管道类,可以同时指定多个管道(以逗号分隔)。在“:”后紧跟的数字表示优先级(表示其执行的顺序),数字越小优先级越高。通常这些数字的取值范围在 0~1000 之内。

```
ITEM_PIPELINES = {
    'chinanews_crawler.pipelines.BlockGamePipeline':300,
    'chinanews_crawler.pipelines.CleanHTMLPipeline':301,
    'chinanews_crawler.pipelines.JsonFeedPipeline':302
}
```

保存 settings.py 后打开命令行窗口运行爬网:

```
$ scrapy crawl chinanews
```

显示的效果如下图所示。



```
(venv) RayOSX:chinanews_crawler Ray$ scrapy crawl chinanews
2017-12-28 13:06:37 [scrapy.utils.log] INFO: Scrapy 1.4.0 started (bot: chinanews_crawler)
2017-12-28 13:06:37 [scrapy.utils.log] INFO: Overridden settings: {'NEWSPIDER_MODULE': 'chinanews_crawler.spiders', 'FEED_FORMAT': 'json', 'SPIDER_MODULES': ['chinanews_crawler.spiders.chinanews'], 'FEED_URI': 'result.json', 'BOT_NAME': 'chinanews_crawler'}
2017-12-28 13:06:37 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.feedexport.FeedExporter',
 'scrapy.extensions.memusage.MemoryUsage',
 'scrapy.extensions.logstats.LogStats',
 'scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.corestats.CoreStats']
2017-12-28 13:06:38 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2017-12-28 13:06:38 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.spidermiddlewares.referer.RefererMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2017-12-28 13:06:38 [scrapy.middleware] INFO: Enabled item pipelines:
['chinanews_crawler.pipelines.BlockGamePipeline',
 'chinanews_crawler.pipelines.CleanHTMLPipeline',
 'chinanews_crawler.pipelines.JsonFeedPipeline']
2017-12-28 13:06:38 [scrapy.core.engine] INFO: Spider opened
2017-12-28 13:06:38 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min)
```

基本配置项

自定义管道

当 Scrapy 启动时，会显示当前由配置文件中加载到引擎运行的实际配置内容。这个命令执行完毕后会有一长串的内容，因为都是一些采集数据的日志输出，在这里就暂且跳过，拉到屏幕的末端我们来看一下 Scrapy 的运行结果，如下图所示。

```
(venv) RayOSX:chinanews_crawler Ray$ scrapy crawl chinanews
2017-12-28 13:06:40 [scrapy.core.engine] DEBUG: Crawled (200) -GET http://www.chinanews.com/rss/edu.xml (referer: http://www.chinanews.com/rss/rss_2.html)
2017-12-28 13:06:40 [scrapy.core.engine] INFO: Closing spider (finished)
2017-12-28 13:06:40 [scrapy.extensions.feedexport] INFO: Stored json feed (531 items) in: result.json
2017-12-28 13:06:40 [scrapy.statscollectors] INFO: Dumping Scrapy stats:
{'downloader/request_bytes': 9128,
 'downloader/request_count': 33,
 'downloader/request_method_count/GET': 33,
 'downloader/response_bytes': 99158,
 'downloader/response_count': 33,
 'downloader/response_status_count/200': 33,
 'finish_reason': 'finished',
 'finish_time': datetime.datetime(2017, 12, 28, 5, 6, 40, 765532),
 'item_dropped_count': 1,
 'item_dropped_reasons_count/DropItem': 1,
 'item_scraped_count': 531,
 'log_count/DEBUG': 565,
 'log_count/ERROR': 8,
 'log_count/INFO': 8,
 'log_count/WARNING': 1,
 'memusage/max': 48889856,
 'memusage/startup': 48889856,
 'request_depth_max': 1,
 'response_received_count': 33,
 'scheduler/dequeued': 33,
 'scheduler/dequeued/memory': 33,
 'scheduler/enqueued': 33,
 'scheduler/enqueued/memory': 33,
 'spider_exceptions/AttributeError': 8,
 'start_time': datetime.datetime(2017, 12, 28, 5, 6, 38, 280547)}
2017-12-28 13:06:40 [scrapy.core.engine] INFO: Spider closed (finished)
(venv) RayOSX:chinanews_crawler Ray$
```

被过滤出来的Item





这里表明了有一个项目被过滤出来。

## 2.6 Scrapy的运行与配置

Scrapy 是一个具有高度扩展性的框架,这种扩展性在很大程度上要依赖于它的配置文件。每个 Scrapy 项目中都会具有一个 `settings.py` 文件,当使用 `scrapy startproject` 指令创建项目时,这个文件就会被一同自动创建。

Scrapy 设定 (settings) 提供了定制 Scrapy 组件的方法,用户可以控制包括核心 (core)、插件 (extension)、pipeline 及 spider 组件。

设定为代码提供了提取以 key-value 映射的配置值的全局命名空间 (namespace)。设定可以通过下面介绍的多种机制进行设置。

设定 (settings) 同时也是选择当前激活的 Scrapy 项目的方法 (如果有多个)。

### 指定设定 (Designating the settings)

当使用 Scrapy 时,需要声明所使用的设定,可以通过使用环境变量 `SCRAPY_SETTINGS_MODULE` 来完成。

`SCRAPY_SETTINGS_MODULE` 必须以 Python 路径语法编写,如 `myproject.settings`。

注意,设定模块应该在 Python import search path 中。

### 获取设定值 (Populating the settings)

设定可以通过多种方式设置,每个方式具有不同的优先级。下面以优先级降序的方式给出方式列表:

- (1) 命令行选项 (Command line Options), 最高优先级。
- (2) 项目设定模块 (Project settings module)。
- (3) 命令默认设定模块 (Default settings per-command)。
- (4) 全局默认设定 (Default global settings), 最低优先级。

这些设定 (settings) 由 Scrapy 内部很好地进行了处理,不过仍可以使用 API 调用来手动处理。

#### ➤ 命令行选项 (Command line options)

命令行传入的参数具有最高优先级,可以使用命令行选项 `-s` (或 `--set`) 来覆盖一个 (或更多) 选项。



样例:

```
$ scrapy crawl myspider -s LOG_FILE=scrapy.log
```

### ➤ 项目设定模块 (Project settings module)

项目设定模块是 Scrapy 项目的标准配置文件，是获取大多数设定的方法。例如，`myproject.settings`。

### ➤ 命令默认设定 (Default settings per-command)

每个 Scrapy tool 命令拥有其默认设定，并覆盖了全局默认的设定。这些设定在命令的类的 `default_settings` 属性中指定。

### ➤ 默认全局设定 (Default global settings)

全局默认设定存储在 `scrapy.settings.default_settings` 模块，并在内置设定参考手册部分有记录。

### ➤ 以编程方式获取配置

设定可以通过 Crawler 的 `scrapy.crawler.Crawler.settings` 属性进行访问。由插件及中间件的 `from_crawler()` 类方法传入：

```
class MyExtension(object):

    @classmethod
    def from_crawler(cls, crawler):
        settings = crawler.settings
        if settings['LOG_ENABLED']:
            print "log is enabled!"
```

Scrapy 的配置项请查看在线 PDF



`DOWNLOADER_MIDDLEWARES_BASE` 的默认值：

```
{
    'scrapy.contrib.downloadermiddleware.robotstxt.RobotsTxtMiddleware': 100,
    'scrapy.contrib.downloadermiddleware.httpauth.HttpAuthMiddleware': 300,
```





```
'scrapy.contrib.downloadermiddleware.downloadtimeout.DownloadTimeoutMiddleware':  
350,  
'scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware': 400,  
'scrapy.contrib.downloadermiddleware.retry.RetryMiddleware': 500,  
'scrapy.contrib.downloadermiddleware.defaultheaders.DefaultHeadersMiddleware':  
550,  
'scrapy.contrib.downloadermiddleware.redirect.MetaRefreshMiddleware': 580,  
'scrapy.contrib.downloadermiddleware.httpcompression.HttpCompressionMiddleware':  
590,  
'scrapy.contrib.downloadermiddleware.redirect.RedirectMiddleware': 600,  
'scrapy.contrib.downloadermiddleware.cookies.CookiesMiddleware': 700,  
'scrapy.contrib.downloadermiddleware.httpproxy.HttpProxyMiddleware': 750,  
'scrapy.contrib.downloadermiddleware.chunked.ChunkedTransferMiddleware': 830,  
'scrapy.contrib.downloadermiddleware.stats.DownloaderStats': 850,  
'scrapy.contrib.downloadermiddleware.httpcache.HttpCacheMiddleware': 900,  
}
```

DOWNLOAD\_HANDLERS\_BASE 的默认值:

```
{  
    'file': 'scrapy.core.downloader.handlers.file.FileDownloadHandler',  
    'http': 'scrapy.core.downloader.handlers.http.HttpDownloadHandler',  
    'https': 'scrapy.core.downloader.handlers.http.HttpDownloadHandler',  
    's3': 'scrapy.core.downloader.handlers.s3.S3DownloadHandler',  
}
```

EXTENSIONS\_BASE 的默认值:

```
{  
    'scrapy.contrib.corestats.CoreStats': 0,  
    'scrapy.webservice.WebService': 0,  
    'scrapy.telnet.TelnetConsole': 0,  
    'scrapy.contrib.memusage.MemoryUsage': 0,  
    'scrapy.contrib.memdebug.MemoryDebugger': 0,  
    'scrapy.contrib.closespider.CloseSpider': 0,  
    'scrapy.contrib.feedexport.FeedExporter': 0,  
    'scrapy.contrib.logstats.LogStats': 0,  
    'scrapy.contrib.spiderstate.SpiderState': 0,  
    'scrapy.contrib.throttle.AutoThrottle': 0,  
}
```

SPIDER\_CONTRACTS\_BASE 的默认值:

```
{
    'scrapy.contracts.default.UrlContract': 1,
    'scrapy.contracts.default.ReturnsContract': 2,
    'scrapy.contracts.default.ScrapesContract': 3,
}
```

SPIDER\_MIDDLEWARES\_BASE 的默认值:

```
{
    'scrapy.contrib.spidermiddleware.httperror.HttpErrorMiddleware': 50,
    'scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware': 500,
    'scrapy.contrib.spidermiddleware.referer.RefererMiddleware': 700,
    'scrapy.contrib.spidermiddleware.urllength.UrlLengthMiddleware': 800,
    'scrapy.contrib.spidermiddleware.depth.DepthMiddleware': 900,
}
```

## 2.7 新闻供稿爬虫的Scrapy实现

本节实例中会采用上一章爬取中新网新闻供稿作为示例的命题，与之不同的是，本示例将基于 Scrapy 实现，目的是将本章中所提及的关于 Scrapy 相关基础内容进行实践演示，通过动手写代码才能快速有效地理解 Scrapy 中各个组成部分的作用与具体用法。

首先，采用 Scrapy 工具命令建立新闻供稿爬虫项目，在命令行状态下键入以下指令：

```
$ scrapy startproject chinanews_cawler
```

成功执行以上指令后，Scrapy 会在当前工作目录中创建以下文件目录结构：

```
.
├── chinanews_crawler
│   ├── chinanews_crawler
│   │   ├── __init__.py
│   │   ├── items.py
│   │   ├── pipelines.py
│   │   ├── settings.py
│   │   └── spiders
```



```

|       └─ __init__.py
└─ scrapy.cfg

```

然后, 按照 2.3 节“数据模型——Item”中提及的构建模型的方法编写一个 NewsFeedItem 类用于承载爬取回来的数据内容, 在~/chinanews\_crawler/chinanews\_crawler/items.py 文件中添加以下代码:

```

# coding:utf8
# chinanews_crawler/chinanews_crawler/items.py
from scrapy.item import Item, Field

class NewsFeedItem(Item):
    title = Field() # 标题
    link = Field() # 新闻详情链接
    desc = Field() # 新闻综述
    pub_date = Field() # 发布日期

```

接下来就可以编写蜘蛛的爬网逻辑代码了, 打开~/chinanews\_crawler/chinanews\_crawler/spiders/\_\_init\_\_.py 并加入以下代码:

```

# chinanews_crawler/chinanews_crawler/spiders/__init__.py
from scrapy.spiders import Spider
from scrapy.http import Request
from ..items import NewsFeedItem
from bs4 import BeautifulSoup

class ChinaNewsSpider(Spider):
    name = "chinanews"
    allow_domain = ["chinanews.com"] # 此项可以不声明
    start_urls = (
        'http://www.chinanews.com/rss/rss_2.html',
    )

    def parse(self, response):
        rss_page = BeautifulSoup(response.body, "html.parser")
        rss_links = set([item['href'] for item in rss_page.find_all('a')])
        for link in rss_links:
            yield Request(url=link, callback=self.parse_feed)

```

```
def parse_feed(self, response):
    rss = BeautifulSoup(response.body, 'lxml')
    for item in rss.find_all('item'):
        feed_item = NewsFeedItem()
        feed_item['title'] = item.title.text
        feed_item['link'] = item.link.text
        feed_item['desc'] = item.description.text
        feed_item['pub_date'] = item.pubdate.text

    yield feed_item
```

2.4 节采用命令参数的方式将爬取的供稿数据输出到 JSON 格式的文件中保存,但这样做命令就会变得很长不便于记忆。我们可以利用 Scrapy 的配置让 Scrapy 默认输出到 JSON 文件,只要在配置文件中加入以下代码即可:

```
FEED_URI = 'result.json'
FEED_FORMAT = 'json'
```

最后在命令行中使用以下指令就可以直接启动爬虫了:

```
(venv) OSX:chinanews_crawler $ scrapy crawl chinanews
```

执行完成后,在 chinanews\_crawler 的项目目录中就会出现一个 result.json 文件,其中保存了所有采集到的新闻数据。

## 2.8 小结

本章以中新网新闻供稿示例来贯穿整个 Scrapy 的基本学习内容,旨在为读者建立对 Scrapy 最基本的认识。相信读者动手写过一次之后,可以开始 Scrapy 的爬网之路了,这也意味着读者已经掌握了初等的虫术,当面对一些结构内容简单的网站上的数据时,应该可以轻易地爬取下来了。

既然本章已经结束,是否意味着 Scrapy 就只有这些内容?答案显然是否定的,前面曾说过 Scrapy 的中文文档极其混乱,而且内容也非常分散。在仔细整理并融入自身的使用经验后,我认为在初始阶段掌握 Scrapy 的 Spider、Item 和 Pipeline 的写法就足够了,而在本书后面的章节中,我仍然会基于 Scrapy 去讲解更多的示例,同时会涉及 Scrapy 更多的高级内容,当读者知道这些模块是用来干什么的时候再记住它要比一开始囫囵吞枣式地死记更加牢固。

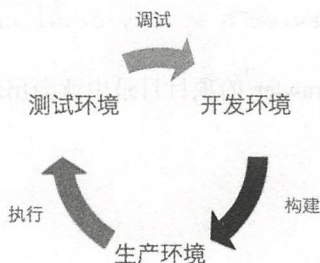


# 3 chapter

## 第 3 章

### Scrapy 的工程管理

无论使用什么语言、何种开发框架，我最重视的就是工程环境。所谓的工程环境，简单来说就是开发环境、测试环境与生产环境，其关系如下图所示。



这三大环境虽然都是在运行工程代码，但其侧重点各有不同，而且理想的效果是不要将它们同置于同一机器之上。首先，开发环境的侧重点是**增加开发效率**，Scrapy 命令行工具就是一套针对爬虫开发与维护的工具。只要细细研读每个指令的作用，会发现它们只不过是將一系列的人工操作通过一个指令一次性地完成罢了，其本质上并没有什么很炫酷的功能。

开发环境工具的最大职责就是要尽可能地将大量重复性的人工操作转变为单一的、自动化的操作。爬虫开发环境相比于 Web 或者移动端开发，是属于非常轻量级的，开发框架 (Scrapy) 本身就提供了这样的能力。如果使用 Python 进行 Web 开发，那么像 Fabric 这样的自动化工具就必不可少，否则只能耗费大量的时间去做那些无趣而耗时的工作。

其次，测试环境的输入正是开发环境的输出，这是保证工程质量的重要一环，相对于其他

系统,爬虫系统是一种逻辑性极其简单的系统,这就意味着它的测试并不会带有过多的复杂性,Scrapy 的 Contracts 功能可以很好地帮助我们为蜘蛛编写专门的测试程序。也就是说,在测试这一环我们仍然可以依赖 Scrapy 命令行工具带来的便利性。

最后,生产环境就是指爬虫项目的最终实际运行环境。生产环境的重要性是与爬虫系统的性质相关的,如果是用于短暂爬取数据的小型爬虫工具,则对于生产环境的要求不高,甚至不进行三大环境的分离,对其本身影响也不大。但对于要支持持久运行、渐进式或增量式爬取数据的爬虫系统而言,生产环境的性能、稳定性及持续交付能力就显得极为重要,尤其是持续交付能力。网站的反爬机制的更新、网页结构的变更、身份验证机制的改变,这些都可能導致我们需要对爬虫的代码进行调整、测试并重新部署。在这种情况下,三大环境的相互配合则会大大地降低在持续迭代与变更中所带来的时间成本与人力成本。

就 Scrapy 框架而言,要做到三大环境分离并不复杂,只需要在原有的 Scrapy 框架中结合 Scrapyd 与 scrapyd-client 工具就可以轻松地构建开发、测试与生产一体化的环境。具体实现逻辑如下:

开发与测试环境都基于 Scrapy,开发环境中以 scrapyd-client 作为自动化部署工具,而生产环境中将以 Scrapyd 工具为宿主,向外部提供生产环境的管理接口。

当然,如果爬虫项目并没有任何迭代需求,或者只需要在单机上短暂地运行来获取一些数据,此时就不必使用 Scrapyd。因为它只对迭代开发和重复部署有帮助。

## 3.1 Scrapyd

Scrapyd 是 Scrapy 官方提供的爬虫管理工具,使用它可以非常方便地上传、控制爬虫及查看运行日志(官方文档参考:<http://scrapyd.readthedocs.org/en/latest/api.html>)。

使用 Scrapyd 和直接运行:

```
$ scrapy crawl myspider
```

有什么区别呢?

Scrapyd 同样是通过上面的命令运行爬虫的,不同的是它提供一个 JSON Web Service 监听请求,我们可以从任何一台能连接到服务器的 PC 发送请求来运行爬虫,或者停止正在运行的爬虫。甚至,我们可以使用它提供的 API 上传新爬虫而不必登录到服务器上进行操作。

可以将 Scrapyd 看作 Scrapy 的一个 Web 包装,通过 Scrapyd 公布的 Restful API 可以让 Scrapy 项目变得具有更好的兼容性,我们就可以使用任何一种语言或工具向 Scrapyd 提供的 Web 地址



发起请求,从而控制 Scrapy 中爬虫的行为。有了 Scrapy 作为 Web 宿主,Scrapy 被部署在网络的任何地方,都可以方便地通过一个 HTTP 的客户端对其进行控制。

除此之外,Scrapy 具有一项特性,十分适合于创建 Scrapy 的生产环境,那就是发行版本控制。Scrapy 的版本控制功能也是通过 Restful API 提供的,这样使得爬虫的迭代与代码的交付变得非常容易。

## 了解 Scrapy

### ➤ 安装 Scrapy

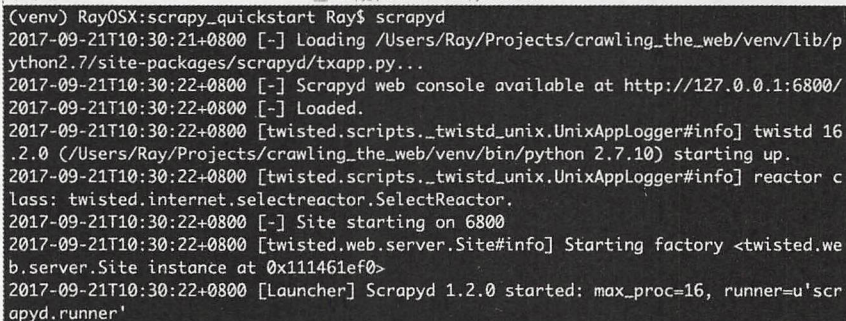
Scrapy 的安装非常简单,通过 pip 就可以完成,具体指令如下所示。

```
$ pip install scrapy
```

### ➤ 运行 Scrapy 服务

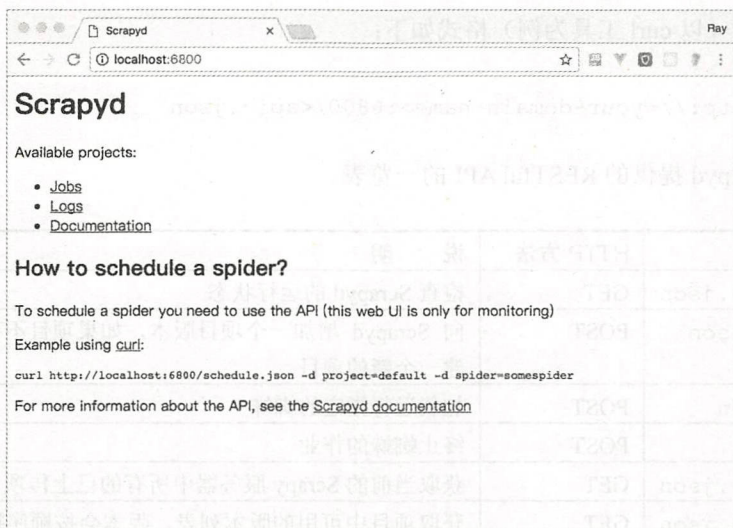
因为 Scrapy 本质上就是一个 Web,所以我们需要通过指令将它加载运行,直接运行命令 scrapy 即可,如下图所示。

```
$ scrapy
```

A terminal window titled 'scrapy\_quickstart -- scrapyd -- 64x16' showing the output of the 'scrapy' command. The output includes the following lines:

```
(venv) RayOSX:scrapy_quickstart Ray$ scrapy
2017-09-21T10:30:21+0800 [-] Loading /Users/Ray/Projects/crawling_the_web/venv/lib/python2.7/site-packages/scrapy/txapp.py...
2017-09-21T10:30:22+0800 [-] Scrapy web console available at http://127.0.0.1:6800/
2017-09-21T10:30:22+0800 [-] Loaded.
2017-09-21T10:30:22+0800 [twisted.scripts._twistd_unix.UnixAppLogger#info] twistd 16.2.0 (/Users/Ray/Projects/crawling_the_web/venv/bin/python 2.7.10) starting up.
2017-09-21T10:30:22+0800 [twisted.scripts._twistd_unix.UnixAppLogger#info] reactor class: twisted.internet.selectreactor.SelectReactor.
2017-09-21T10:30:22+0800 [-] Site starting on 6800
2017-09-21T10:30:22+0800 [twisted.web.server.Site#info] Starting factory <twisted.web.server.Site instance at 0x111461ef0>
2017-09-21T10:30:22+0800 [Launcher] Scrapy 1.2.0 started: max_proc=16, runner='scrapy.runner'
```

默认情况下 Scrapy 监听 0.0.0.0:6800 端口。运行 Scrapy 后,在浏览器中输入 <http://localhost:6800/> 中即可查看当前可以运行的项目,如下图所示。



如果要在 Ubuntu 上安装 Scrapy，则使用 `$ apt-get install scrapy` 指令可以直接将 Scrapy 安装到系统根目录中。

现在 Scrapy 已经成功启用，因为它是作为一个 Master 端存在的，所以使用时不要退出 Scrapy 的命令窗口。当然，如果将 Scrapy 作为生产环境的 Web 宿主，我们还得将 Scrapy 配置为系统服务，在机器启动时就自动加载运行。如果要在外部网络访问，则还得通过 Nginx 将 Scrapy 的监听端口进行映射，这部分内容会在本章的最后进行讲解。

## 工程的部署

要完整地将项目部署到服务器上，需要通过 Scrapy 的 `addversion.json` 服务将 Python 编译器生成 `egg` 文件（也就是 Python 独有的包描述信息）并上传到服务器。我们可以以手工方式将文件上传到服务器，但最简单的办法是通过 `scrapy-deploy` 软件工具包中附带的 `scrapy-client` 客户端来完成这一项工作。

## RESTful API

Scrapy 的 Web 界面比较简单，主要用于监控，所有的调度工作全部依靠 Web 接口实现。也就是说，通过任意可以发出 HTTP 请求的工具（如 `urllib`）向 Scrapy 服务器指定的地址发出请求以获得具体的服务。如果在命令行中调用 Scrapy 上的功能，可以通过 `curl` 工具来实现。

官方文档：<http://scrapy.readthedocs.org/en/stable/api.html>。

Scrapy 的指令方式都非常容易记忆，每个“Web”方法都对应一个其虚拟根目录下的 `*.json`



文件。具体调用（以 curl 工具为例）格式如下：

```
$ curl http://<your-domain-name>:6800/<api>.json
```

以下是 Scrapyd 提供的 RESTful API 的一览表。

URI	HTTP 方法	说 明
daemonstatus.json	GET	检查 Scrapyd 的运行状态
addversion.json	POST	向 Scrapyd 增加一个项目版本，如果项目不存在，则自动创建一个新项目
schedule.json	POST	加载运行指定的蜘蛛
cancel.json	POST	终止蜘蛛的作业
listprojects.json	GET	获取当前的 Scrapy 服务器中所有的已上传项目列表
listversions.json	GET	获取项目中可用的版本列表。版本会按顺序排列，最后一个为当前正在使用的版本
listspiders.json	GET	获取当前版本中可用的蜘蛛列表
listjobs.json	GET	获取项目中待定、正在运行或者已完成的作业列表
delversion.json	POST	删除项目中指定的版本
delproject.json	POST	删除指定项目及其所有已上传的版本

从上表就可以了解 Scrapyd 到底可以为我们做些什么了。接下来对上表中的指令用法进行具体的分类与叙述。从功能上划分可以将以上 API 分为查询、版本管理和作业管理三类。

## 查询API

### ➤ daemonstatus.json

检查 Scrapyd 的运行状态。

指令：

```
$ curl http://localhost:6800/daemonstatus.json
```

返回结果：

```
{
  "status": "ok",
  "running": "0",
  "pending": "0",
```

```
"finished": "0",  
"node_name": "ubuntu"  
}
```

#### ➤ listprojects

获取当前的 Scrapy 服务器中所有的已上传项目列表。

指令:

```
$ curl http://localhost:6800/listprojects.json
```

返回结果:

```
{  
  "status": "ok",  
  "projects": ["myproject", "otherproject"]  
}
```

#### ➤ delproject.json

删除指定项目及其所有已上传的版本。

指令:

```
$ curl http://localhost:6800/delproject.json -d project=tutorial
```

参数:

- project (string, 必选)——项目名称。

返回结果:

```
{  
  "status": "ok"  
}
```

### 版本管理API

#### ➤ listversions.json

获取项目中可用的版本列表。版本会按顺序排列最后一个为当前正在使用的版本。

指令:



```
$ curl http://localhost:6800/listversions.json?project=myproject
```

返回结果:

```
{
  "status": "ok",
  "versions": ["r99", "r156"]
}
```

#### ➤ addversion.json

向 Scrapyd 增加一个项目版本, 如果项目不存在, 则自动创建一个新的项目。

指令:

```
$ curl http://localhost:6800/addversion.json -F project=myproject -F
version=r23 -F egg=@myproject.egg
```

参数:

- project (string, 必选)——项目名称;
- version (string, 必选)——项目版本;
- egg (file, required)——Python 项目的 Egg 信息。

返回结果:

```
{
  "status": "ok",
  "spiders": 3
}
```

#### ➤ delversion.json

删除项目中指定的版本。

指令:

```
$ curl http://localhost:6800/delversion.json -d project=myproject -d
version=r99
```

参数:

- project (string, 必选)——项目名称;
- version (string, 必选)——项目版本。

返回结果:

```
{
  "status": "ok"
}
```

## 作业管理API

### ➤ listspiders.json

获取当前版本中可用的蜘蛛列表。

```
$ curl http://localhost:6800/listspiders.json?project=tutorial_deploy
```

参数:

- project (string, 必选)——项目名称;
- \_version (string, 可选)——指定版本号。

返回结果:

```
{
  "status": "ok",
  "spiders": ["spider1", "spider2", "spider3"]
}
```

### ➤ listjobs.json

获取项目中待定、正在运行或已完成的作业列表。

```
$ curl http://localhost:6800/listjobs.json?project=myproject
```

参数:

- project (string, 必选)——项目名称。

返回结果:

```
{
  "status": "ok",
```



```
{
  "pending": [{
    "id": "78391cc0fcafl1e1b0090800272a6d06",
    "spider": "spider1"
  }],
  "running": [{
    "id": "422e608f9f28cef127b3d5ef93fe9399",
    "spider": "spider2",
    "start_time": "2012-09-12 10:14:03.594664"
  }],
  "finished": [{
    "id": "2f16646cfcafl1e1b0090800272a6d06",
    "spider": "spider3",
    "start_time": "2012-09-12 10:14:03.594664",
    "end_time": "2012-09-12 10:24:03.594664"
  }]
}
```

#### ➤ cancel.json

终止蜘蛛的作业。

指令:

```
$ curl http://localhost:6800/cancel.json -d project=tutorial -d
job=94bd8ce041fd11e6af1a000c2969bafd
```

参数:

- project (string, 必选)——项目名称;
- job (string, 必选)——作业编号。

返回结果:

```
{
  "status": "ok",
  "prevstate": "running"
}
```

#### ➤ schedule.json

加载运行指定的蜘蛛。

指令:

```
$ curl http://localhost:6800/schedule.json -d project=tutorial -d spider=tencent
```

参数:

- project (string, 必选) —— 项目名称;
- spider (string, 必选) —— 蜘蛛名称;
- setting, string, 可选) —— 声明的设置项目, 可用于覆盖 Scrapy 的默认运行设定;
- jobid (string, 可选) —— 直接指定一个具体的工作 ID, 用于取代默认生成的 UUID;
- \_version (string, 可选) —— 指定运行该项目中的某个具体版本;
- 传入其他参数则直接作为蜘蛛的运行参数。

返回结果:

```
{
  "status": "ok",
  "jobid": "94bd8ce041fd11e6af1a000c2969bafd",
  "node_name": "ubuntu"
}
```

## 配置

Scrapyd 会按照以下序列从上至下读取运行配置, 序数越大的优先级越高。也就是说, 具有相同项目时, 优先级越高的文件配置会覆盖优先级别低的文件配置:

- (1) /etc/scrapyd/scrapyd.conf (UNIX)。
- (2) c:\scrapyd\scrapyd.conf (Windows)。
- (3) /etc/scrapyd/conf.d/\* (UNIX)。
- (4) scrapyd.conf。
- (5) ~/.scrapyd.conf (用户目录)。

配置项与说明如下表所示。

配置项	说 明
http_port	Restful API 的监听端口, 默认为 6800
bind_address	Restful API 宿主服务的 IP 地址, 默认为 127.0.0.1 (localhost)



续表

配置项	说明
max_proc	Scrapy 并发进程的最大数量。如果不设置或者设置为 0，则采用当前系统中 max_proc_per_cpu 选项设定的可用 CPU 数为基准，默认为 0
max_proc_per_cpu	每个 CPU 可以启动的最大并发 Scrapy 进程数，默认为 4
debug	是否启用调试模式。默认为关闭。当启用调试模式时，一旦处理 JSON API 调用时出现错误，将返回完整的 Python 跟踪信息（纯文本形式）
eggs_dir	Python 编译后生成的 eggs 文件存放目录
dbs_dir	项目数据库的存放目录（包括蜘蛛的队列）
logs_dir	日志目录。设置为空时可禁止写入日志
items_dir	存放爬取数据的目录。此选项被默认设置为不可用状态，因为通常我们会将爬取的数据保存数据库或者数据导出文件中。如果设置该选项，则将取代 Scrapy 设置中的 FEED_URI 配置（在数据存储一节中会有详细解释）
jobs_to_keep	保持每个蜘蛛完成的工作数量，默认为 5
finished_to_keep	保存在启动器中的完成进程的数量，默认为 100。仅用于设定网站的服务进程
poll_interval	用于轮询队列的时间间隔，以秒为单位。默认为 5.0，可以为小数，例如，0.2 用于启动子进程模块，这样就可以使用自己的模块来自定义从 Scrapy 启动的 Scrapy 进程
application	返回（Twisted）应用程序对象的函数。可以返回一个扩展 Scrapy 的 Web 应用程序对象以增加或移取默认提供的组件或服务
node_name	设置每个节点的名称，类似于主机名。默认为 \${socket.gethostname() }

以下是一个详细的 Scrapy 配置的例子：

```
[scrapy]
eggs_dir = eggs
logs_dir = logs
items_dir =
jobs_to_keep = 5
dbs_dir = dbs
max_proc = 0
max_proc_per_cpu = 4
finished_to_keep = 100
poll_interval = 5.0
bind_address = 127.0.0.1
http_port = 6800
```

```

debug = off
runner = scrapy.runner
application = scrapyd.app.application
launcher = scrapyd.launcher.Launcher
webroot = scrapyd.website.Root

[services]
schedule.json = scrapyd.webservice.Schedule
cancel.json = scrapyd.webservice.Cancel
addversion.json = scrapyd.webservice.AddVersion
listprojects.json = scrapyd.webservice.ListProjects
listversions.json = scrapyd.webservice.ListVersions
listspiders.json = scrapyd.webservice.ListSpiders
delproject.json = scrapyd.webservice.Schedule
delversion.json = scrapyd.webservice.Cancel
listjobs.json = scrapyd.webservice.AddVersion
daemonstatus.json = scrapyd.webservice.DaemonStatus

```

## 3.2 scrapyd-client及部署

Scrapyd-client (<https://github.com/scrapy/scrapyd-client>) 是 Scrapy 的一个客户端工具，它提供了一系列的指令工具以简化 Scrapy 那些冗长的 HTTP 指令请求，另外它还搭载了 scrapyd-deploy 部署工具，可以将开发环境中的代码部署到远程的 Scrapy 服务器中。

### 安装

默认的安装方法：

```
$ pip install scrapyd-client
```

scrapyd-client 的官方版本在安装完成后是没有 scrapyd-client 指令的，启用这个指令需要直接安装 master branch 上的版本：

```
$ pip install --upgrade git+ssh://git@github.com/scrapy/scrapyd-client.git
```

指令用法：



```
$ scrapyd-client [-h] [-t TARGET] {deploy,projects,schedule,spiders} ...
```

所有 scrapyd-client 下的子指令集都可以通过 `$ scrapyd-client <指令> --help` 方式获取具体的帮助信息。

scrapyd-client 主要提供以下几个指令集:

- projects
- schedule
- spiders
- deploy

#### ➤ projects

列出 Scrapy 实例中的所有项目:

```
# 列出默认服务器上的所有项目
```

```
$ scrapyd-client projects
```

```
# 从指定的 Scrapy 服务器地址上列出所有的项目
```

```
scrapyd-client -t http://scrapy.example.net projects
```

#### ➤ schedule

启动一个或多个蜘蛛的爬网任务:

```
# 启动任意的蜘蛛
```

```
$ scrapyd-client schedule
```

```
# 启动指定项目 (sina) 中的所有蜘蛛
```

```
$ scrapyd-client schedule -p sina *
```

```
# 支持通配符, 启动所有以 _daily 结尾的项目中的蜘蛛
```

```
scrapyd-client schedule -p * *_daily
```

#### ➤ spiders

列出项目中的蜘蛛:

```
# 列举出所有的蜘蛛
```

```
$ scrapyd-client spiders

# 列出指定项目(sina)下的蜘蛛
$ scrapyd-client spiders -p sina
```

### ➤ delpoy

将本机上的项目部署到 Scrapyd 服务器上, 由于这个指令是 scrapyd-client 最常用的, 因此 scrapyd-client 还提供了一个 scrapy-delpoy 的指令, scrapy-delpoy 的使用效果与 scrapyd-client deploy 是一样的。接下来就重点讲述 scrapy-delpoy 的使用方法。

## 部署Scrapy项目

通常将本机上的项目部署到 Scrapyd 服务器上需要执行以下两个步骤:

- 将当前项目中的 egg 信息发布到服务器中(需要在当前项目下安装 setuptools 以支持此操作);
- 通过调用 addversion.json API 将 egg 上传到 Scrapyd 服务器。

scrapyd-deploy 就是为自动完成以上两个步骤而制作的。

首先进入项目的根目录, 然后按照以下格式使用 scrapyd-deploy 指令:

```
$ scrapyd-deploy <target> -p <project>
```

当 scrapyd-deploy 执行后, 会将项目的 egg 信息上传到服务器, 这个过程中 scrapyd-deploy 会使用 setuptools 对本地项目进行打包, 因此 scrapyd-deploy 会查询当前项目的根目录内是否已存有 setup.py 文件, 如果没有, 则自动创建一份。

当执行成功后会显示类似以下 JSON 格式的信息:

```
Deploying myproject-1287453519 to http://localhost:6800/addversion.json
Server response (200):
{"status": "ok", "spiders": ["spider1", "spider2"]}
```

我们将远程 Scrapyd 服务器的地址保存到 Scrapy 的配置文件 (scrapy.cfg) 中, 这样每次发布项目时就不需要重复输入服务器地址了:

```
[deploy:sina] #默认情况下并没有 scrapyd2, 它只是一个名字, 可以在配置文件中写多个名字不同的 deploy
```



```
url = http://localhost:6800 #要部署项目的服务器的地址
username = ray #访问服务器所需的用户名和密码（如果不需要则密码可以不写）
password = secret
```

其中的 `username` 和 `password` 用于在部署时验证服务器的 HTTP basic authentication, 需要注意的是, 这里的用户密码并不表示访问该项目时需要验证, 而是登录服务器用的。

写入配置后就可以直接执行部署指令了:

```
$ scrapyd-deploy
```

如果在配置中设置了多个目标服务器地址, 则可以用以下指令一次性将项目部署到各个服务器上:

```
$ scrapyd-deploy -a -p
```

## 版本管理

默认情况下, `scrapyd-deploy` 会采用当前的时间戳来生成项目的版本号。也可以用 `--version` 来设置自定义的版本号:

```
$ scrapyd-deploy -p --version
```

使用 `-a` 参数应用到所有的目标服务器:

```
$ scrapyd-deploy -a -p --version
```

当没有声明具体版本号时, Scrapy 会自动取得一个最适用的版本值。

如果使用 Mercurial 或 Git, 则可以用 `HG` 或者 `GIT` 分别作为 `--version` 的参数值并保存到 `scrapy.cfg` 文件中:

```
[deploy:target]
...
version = HG
```

关于 `_LooseVersion` 类的详细说明可以参考: <http://epydoc.sourceforge.net/stdlib/distutils.version.LooseVersion-class.html>。

## 关于egg的注意事项

在部署前构建 egg 信息时需要注意以下几点。

- 确认在 egg 信息中没有包含本地开发设置。在绝大多数情况下我们只会上传 egg 的默认设置, 比如 `find_packages` 方法会获取本地机器上的自定义设置, 因此不建议采用。
- 在代码中应该避免使用 `__file__` 变量, 因为在生成 egg 时就会包含本地信息, 一旦上传到服务器就可能失效。可以采用 `pkgutil.get_data` 方法将其取代。
- 要对所有的写盘操作格外小心, 因为 Scrapyd 会以 Web 形式运行, 这就意味着项目运行于一个多用户环境之中。也就是说, 并不是所有用户都具有写盘的权限, 因此必须确保启动进程有写盘权限, 或者将写入的数据保存到临时文件 (`_template`) 中。
- `_pkgutil.get_data`: 具体参考 [http://docs.python.org/library/pkgutil.html#pkgutil.get\\_data](http://docs.python.org/library/pkgutil.html#pkgutil.get_data)。
- `_tempfile`: 具体参考 <http://docs.python.org/library/tempfile.html>。

## 部署目标

可以在 `scrapy.cfg` 文件中定义多个部署目标, 例如:

```
[deploy:example]
url = http://scrapy.example.com/api/scrapy
username = scrapy
password = secret
```

通过 `-l` 参数可以查询当前项目中全部的部署目标, 例如:

```
$ scrapy-deploy -l
```

也可以通过声明具体项目名称查看其具体的部署目标 `-L`, 具体如下:

```
$ scrapy-deploy -L example
```

## 3.3 搭建爬虫服务器

首先, 我们希望 Scrapyd 可以像系统服务一样在操作系统启动后自动启动并运行, 那么我们就需要使用 Systemd 来完成这一系统引导工作。Systemd 是 Linux 系统工具, 用来启动守护进程, 已成为大多数发行版的标准配置。



先检查系统中是否安装 Systemd 及确定当前安装的版本:

```
$ systemd --version
$ sudo vi /lib/systemd/system/scrapyd.service
```

然后将以下内容添加到 scrapyd.service 文件中:

```
[Unit]
Description=scrapyd
After=network.target
Documentation=http://scrapyd.readthedocs.org/en/latest/api.html

[Service]
User=root
ExecStart=/usr/local/bin/scrapyd --logfile /var/scrapyd/scrapyd.log

[Install]
WantedBy=multi-user.target
```

- [Unit] 区块通常是配置文件的第一个区块, 用来定义 Unit 的元数据, 以及配置与其他 Unit 的关系。
- After: 如果该字段指定的 Unit After 也要启动, 那么必须在当前 service 之前启动。
- Documentation: 服务文档地址。
- Description: 简短描述。
- [Service] 区块用来 Service 的配置, 只有 Service 类型的 Unit 才有这个区块。
- ExecStart: 启动当前服务的命令。
- [Install]: 通常是配置文件的最后一个区块, 用来定义如何启动, 以及是否开机启动。
- WantedBy: 它的值是一个或多个 Target, 当前 Unit 激活时 (enable) 符号链接会放入 /etc/systemd/system 目录下以 Target 名+.wants 后缀构成的子目录中, 由此我们就可以通过命令行启动一个新的服务了。

```
$ sudo systemctl start scrapyd
$ sudo service scrapyd start
```

可以通过以下指令来检查服务状态：

```
$ sudo systemctl status scrapyd
```

通过以下指令让 Scrapyd 随同操作系统一同启动：

```
$ sudo systemctl enable scrapyd
```

最后创建一个 symlink, 从 /etc/systemd/system/multi-user.target.wants/scrapyd.service 链接到 /lib/systemd/system/scrapyd.service。

如果要取消开机启动, 则可以采用以下指令：

```
$ sudo systemctl disable scrapyd
```

### Scrapyd服务器添加认证信息

我们也可以在 Scrapyd 前面加一层反向代理来实现用户认证 (以 Nginx 为例)。

安装 Nginx:

```
$ sudo apt-get install nginx
```

配置 Nginx:

```
$ vi /etc/nginx/nginx.conf
```

修改如下:

```
# Scrapyd local proxy for basic authentication.
# Don't forget iptables rule.
# iptables -A INPUT -p tcp --destination-port 6800 -s ! 127.0.0.1 -j DROP

http {
    server {
        listen 6801;
        location / {
            proxy_pass      http://127.0.0.1:6800/;
            auth_basic       "Restricted";
            auth_basic_user_file /etc/nginx/conf.d/.htpasswd;
```



```

    }
}
}

```

/etc/nginx/htpasswd/user.htpasswd 中设置的用户名和密码都是 test。下面修改配置文件并添加用户信息。

Nginx 使用 htpasswd 创建用户认证:

```

python@ubuntu:/etc/nginx/conf.d$ sudo htpasswd -c .htpasswd ray
New password:
Re-type new password:
Adding password for user ray
python@ubuntu:/etc/nginx/conf.d$ cat .htpasswd
ray:$apr1$2slPhvee$6cqtraHxoxclqf1DpqIPM.

```

```

python@ubuntu:/etc/nginx/conf.d$ sudo htpasswd -bc .htpasswd admin admin

```

Apache htpasswd 命令用法实例

1. 如何利用 htpasswd 命令添加用户?

```
htpasswd -bc .passwd www.dotnetage.com php
```

在 bin 目录下生成一个 .passwd 文件, 用户名为 www.dotnetage.com, 密码为 php, 默认采用 MD5 加密方式。

2. 如何在原有密码文件中增加下一个用户?

```
htpasswd -b .passwd dotnetage phpdev
```

去掉 c 选项, 即可在第一个用户之后添加第二个用户, 以此类推。

重启 Nginx:

```
$ sudo service nginx restart
```

测试 Nginx:

```

F:\____gitProject____\curl-7.33.0-win64-ssl-sspi\tieba_baidu>curl
http://localhost:6800/schedule.json -d project=tutorial -d spider=tencent -u
ray:test
{"status": "ok", "jobid": "5ee61b08428611e6af1a000c2969bafd", "node_name":
"ubuntu"}

```

配置 scrapy.cfg 文件:

```
[deploy]
url = http://192.168.19.12:6801/
project = tutorial
username = admin
password = admin
```

注意上面的 URL 已经修改为 Nginx 监听的端口。

**提醒:** 记得修改服务器上 Scrapyd 的配置 bind\_address 字段为 127.0.0.1, 以免可以从外面绕过 Nginx, 直接访问 6800 端口。关于配置可以参看本文后面的配置文件设置。

修改配置文件:

```
$ sudo vi /etc/scrapyd/scrapyd.conf
```

```
[scrapyd]
bind_address = 127.0.0.1
```

Scrapyd 启动时会自动搜索配置文件, 配置文件的加载顺序为:

```
$ /etc/scrapyd/scrapyd.conf /etc/scrapyd/conf.d/* scrapyd.conf ~/.scrapyd.conf
```

最后加载的配置文件会覆盖前面的配置文件, 默认配置文件如下, 可以根据需要修改。

```
[scrapyd]
eggs_dir      = eggs
logs_dir      = logs
items_dir     = items
jobs_to_keep  = 5
dbs_dir       = dbs
max_proc      = 0
max_proc_per_cpu = 4
finished_to_keep = 100
poll_interval = 5
bind_address  = 0.0.0.0
http_port     = 6800
```



```
debug          = off
runner         = scrapyd.runner
application    = scrapyd.app.application
launcher       = scrapyd.launcher.Launcher

[services]
schedule.json  = scrapyd.webservice.Schedule
cancel.json    = scrapyd.webservice.Cancel
addversion.json = scrapyd.webservice.AddVersion
listprojects.json = scrapyd.webservice.ListProjects
listversions.json = scrapyd.webservice.ListVersions
listspiders.json = scrapyd.webservice.ListSpiders
delproject.json = scrapyd.webservice.DeleteProject
delversion.json = scrapyd.webservice.DeleteVersion
listjobs.json  = scrapyd.webservice.ListJobs
```

当我们完成了爬虫服务器的搭建以后，就一边开发一边部署了。这对于某些需要渐进式开发的爬虫系统而言非常重要，它使得新旧版本程序的部署变得非常简单且可控，即便部署失败还能进行回滚还原。这样一来在开发机上能运行而在实际环境上安装后却整体哑火的情况大大减少，至少程序上线不会再是一场噩梦。

# 4 chapter

## 第 4 章 中阶虫术

前 3 章介绍了虫术的一些基本概念和基本工具的使用，旨在让读者建立基本的思维框架并通过一些简单的例子引发读者对虫术的兴趣。本章会进入另一个领域，那就是从开发一个简单的爬虫程序进阶到开发一个爬虫系统。对一个系统来说，需要了解的技术内容与细节会更多，尤其对我们手上的工具和编程框架需要有深度的认知与理解。

本章将围绕以下几个重点展开深度的探索。

### 蜘蛛的演化

深入 Scrapy 蜘蛛的内部实现代码，全面理解蜘蛛的实现，在学习 Scrapy 搭载的内置蜘蛛的作用与用法的基础上，进一步掌握如何按照项目的实际需求来编写自定义的蜘蛛，最终通过蜘蛛中间件对蜘蛛的行为进行灵活的控制与扩展。

### 如何进行爬虫系统的调试与测试

在开发期间，我们需要知道蜘蛛的爬网逻辑是否符合设计要求。在很多情况下，我们需要了解如何在适当的地方“打断”爬虫系统的运行，观察或调整运行期间的某些变量；当蜘蛛部署到运行环境时，更需要了解其内部的运行状态是否符合预期，有没有出现意想不到的异常。这都需要一些有效的调试/测试的方法与工具。

### 从HTTP协议入手了解蜘蛛对数据的处理方法

从本质上讲，爬虫系统就是一个具有并发行为的 HTTP 客户端，我们有必要深入 HTTP 协议的内部去了解 Scrapy 是如何控制请求的产生，以及如何读取响应的内容。



## 处理JavaScript

在“体验为王”的后互联网时代背景下，如果爬虫不具备对 JavaScript 的处理能力，则对于各种应用了大量前端技术的网站来说会束手无策。怎么才能让蜘蛛具有如浏览器般强大的 JavaScript 处理能力，真实地还原网页在最终用户面前所显的全貌，是本章重点讲述的内容。

## 如何对爬取的数据进行存储

如果说爬虫是大数据生态中的源头与基础，那么数据存储则是大数据生态的基石。爬虫系统既要会“爬”，也要懂得如何去“存”。在爬虫系统强大的采集功能下，产生的海量的数据是可以预估的。大多数情况下我们并不会将这些采集后的样本数据存在本地，而是选择更高速可靠的云存储方案，本章讲解如何基于 Scrapy 实现各种数据存储方式。

## 4.1 蜘蛛的演化

爬虫（Crawler）系统的核心就是“蜘蛛”（Spider），很多情况下两者指的是同一类东西。如果真的要概念细分，则爬虫（Crawler）更多指的是整个系统，而“蜘蛛”（Spider）是爬虫系统中的核心成员/模块。

在爬虫系统的设计与开发中，绝大多数的时间与成本都耗费在蜘蛛的设计上，尤其当我们选定了像 Scrapy 这样成熟的框架作为基础时，就需要深入理解 Scrapy 搭载的原生蜘蛛，甚至更进一步去了解蜘蛛的高层次抽象类的设计，这样才能全面地掌握开发蜘蛛的各种思路与方法。

由于爬取的数据目标各异，所以爬虫内的逻辑是很难被抽象和重用的，但是这并不意味着蜘蛛就没有重用性。蜘蛛的爬取逻辑几乎不可重用，但是蜘蛛的爬取行为模式和目标内容结构方面却有相似性。

本章将详细地探索蜘蛛在内容分析与爬取行为上的各种共性，与此同时，还将学习 Scrapy 在这类共性上所提供的内置蜘蛛能如何帮助我们加速蜘蛛的设计与开发。最后深入蜘蛛的内部了解它的实现原理，这样才能有助于我们应对各种数据爬取场合。

### 4.1.1 蜘蛛的本质——深入Spider

学习一个框架最直接的办法就是查看其顶层的接口或抽象类结构，因为接口与基类从总体上界定了整类库或类族的“特性”与“能力”，这是我二十年来使用面向对象方法学习类库的一点心得。因此，如果想充分了解蜘蛛的运作机理，就应该好好地了解 Spider 基类到底提供了哪些特性与能力，以及它自身是如何运作的。

Spider 类提供了蜘蛛的最基本的行为与特性，其他蜘蛛都必须继承自该类（包括 Scrapy

自带的蜘蛛及用户自己编写的蜘蛛)。Spider 类并没有提供太多特殊的功能，其仅仅请求给定的 `start_urls/start_requests`，并根据返回的结果（resulting responses）然后调用 `parse` 方法对返回结果进行深入爬取或提取出目标数据。

这里不得不先为 Python 点个赞，因为所有的 Python 代码都是开源的，我们很容易就可以查看 Spider 类的源代码是怎么写的，毕竟阅读代码就是学习捷径！接下来打开这个类的源码以探究其中的奥妙：

```
class Spider(object_ref):
    name = None
    custom_settings = None

    def __init__(self, name=None, **kwargs):
        if name is not None:
            self.name = name
        elif not getattr(self, 'name', None):
            raise ValueError("%s must have a name" % type(self).__name__)
        self.__dict__.update(kwargs)
        if not hasattr(self, 'start_urls'):
            self.start_urls = []

    @property
    def logger(self):
        logger = logging.getLogger(self.name)
        return logging.LoggerAdapter(logger, {'spider': self})

    def log(self, message, level=logging.DEBUG, **kw):
        self.logger.log(level, message, **kw)

    @classmethod
    def from_crawler(cls, crawler, *args, **kwargs):
        spider = cls(*args, **kwargs)
        spider._set_crawler(crawler)
        return spider

    def set_crawler(self, crawler):
        warnings.warn("set_crawler is deprecated, instantiate and bound the "
```



```

        "spider to this crawler with from_crawler method "
        "instead.",
        category=ScrapyDeprecationWarning, stacklevel=2)
    assert not hasattr(self, 'crawler'), "Spider already bounded to a " \
        "crawler"

    self._set_crawler(crawler)

def _set_crawler(self, crawler):
    self.crawler = crawler
    self.settings = crawler.settings
    crawler.signals.connect(self.close, signals.spider_closed)

def start_requests(self):
    for url in self.start_urls:
        yield self.make_requests_from_url(url)

def make_requests_from_url(self, url):
    return Request(url, dont_filter=True)

def parse(self, response):
    raise NotImplementedError

@classmethod
def update_settings(cls, settings):
    settings.setdefault(cls.custom_settings or {}, priority='spider')

@classmethod
def handles_request(cls, request):
    return url_is_from_spider(request.url, cls)

@staticmethod
def close(spider, reason):
    closed = getattr(spider, 'closed', None)
    if callable(closed):
        return closed(reason)

def __str__(self):
    return "<%s %r at 0x%0x>" % (type(self).__name__, self.name, id(self))

```

```
__repr__ = __str__
```

从 Spider 类的源代码中可以得知,一旦 Spider 的子类被实例化, `__init__` 中的代码就会被执行:

```
def __init__(self, name=None, **kwargs):
    if name is not None:
        self.name = name
    elif not getattr(self, 'name', None):
        raise ValueError("%s must have a name" % type(self).__name__)
    self.__dict__.update(kwargs)
    if not hasattr(self, 'start_urls'):
        self.start_urls = []
```

那么就需要设定 `name` 与 `start_urls` 两个属性,这就解释了为什么上例中的 `ChinanewsSpider` 要在一开始就初始化 `name` 与 `start_urls` 两个属性。然而 Spider 的子类被实例化后并不会马上执行爬网,只有 `start_requests` 被调用时,蜘蛛才会执行爬网的动作。我们重点看一下 `start_requests` 方法:

```
def start_requests(self):
    for url in self.start_urls:
        yield self.make_requests_from_url(url)
```

这个函数非常简单,它只是一个生成 `make_requests_from_url` 方法调用结果的枚举器,枚举的条件就是 Spider 实例化时设定的 `start_urls`,也就是爬虫爬网的“起点”。那么 `make_requests_from_url` 又干了什么呢?以下是它的代码:

```
def make_requests_from_url(self, url):
    return request(url, dont_filter=True)
```

它只是一个工厂方法,用于生成 `request` 对象。将 `start_urls`、`start_requests` 和 `make_requests_from_url` 合起来理解就是: `start_requests` 一旦被调用就会产生与 `start_urls` 相同数量的 `request` 对象的枚举器。

为什么需要了解这个工作过程?因为只有了解了 `request` 是如何产生的,我们才能在一些特殊的场合深度定制自己的蜘蛛。例如,在很多情况下 `start_urls` 有可能不是一个常量,而是一个生成规则,这样就可以通过方法重写来重新实现 `start_requests`。



如果想修改最初爬取某个网站的 request 对象, 则可以重写 (override) start\_requests 方法。例如, 如果需要在启动时以 POST 登录某个网站, 则可以这么写:

```
def start_requests(self):
    return [scrapy.FormRequest("http://www.example.com/login",
                               formdata={'user': 'john', 'pass': 'secret'},
                               callback=self.logged_in)]

def logged_in(self, response):
    # here you would extract links to follow and return Requests for
    # each of them, with another callback
    pass
    make_requests_from_url(url)
```

该方法接收一个 URL 并返回用于爬取的 request 对象。该方法在初始化 request 时被 start\_requests() 调用, 也被用于转化 url 为 request。

在默认未被复写 (overridden) 的情况下, 该方法返回的 request 对象中, parse() 作为回调函数, dont\_filter 参数也被设置为开启。

我们来看一下 Spider 的另一个重要的方法 parse:

```
def parse(self, response):
    raise NotImplementedError
```

这个方法只会抛出一个“尚未实现”的异常, 这是一个非常好的保护! 这样做可以防止我们在继承 Spider 后“忘记”去重写这个重要接口方法, 因为它是被 Downloader (下载器) 在发出 request 并获取到响应后调用的方法接口。

**注意:** 当 parse 方法中返回 request 对象时, Scrapy 会重新将 request 发向下载器, 待下载器返回响应后重新回到当前的 parse 方法, 或者执行 request 构造函数内传入的回调函数。

只有当 parse 返回的 item 是被爬取的数据, 才会“流动”到下一个环节进行处理。

附: Spider 使用参考 (见下表)

包: class scrapy.spider.Spider

属 性	说 明
name	定义 Spider 名字的字符串(string)。Spider 的名字定义了 Scrapy 如何定位(并初始化) Spider, 所以其必须是唯一的。不过用户可以生成多个相同的 Spider 实例(instance), 这没有任何限制。name 是 Spider 中最重要的属性, 而且是必需的。如果该 Spider 爬取单个网站(single domain), 一个常见的做法是以该网站(domain)(加或不加后缀)来命名 Spider。例如, 如果 Spider 爬取 mywebsite.com, 则该 Spider 通常会被命名为 mywebsite
allowed_domains	可选。包含了 Spider 允许爬取的域名(domain)列表(list)。当 OffsiteMiddleware 启用时, 域名不在列表中的 URL 不会被跟进
start_urls	URL 列表。当没有制定特定的 URL 时, Spider 将从该列表中开始进行爬取。因此, 第一个被获取的页面的 URL 将是该列表之一。后续的 URL 将会从获取的数据中提取
start_requests()	该方法必须返回一个可迭代对象(iterable)。该对象包含了 Spider 用于爬取的第一个 request。当 Spider 启动爬取且未制定 URL 时, 该方法被调用。当指定了 URL 时, make_requests_from_url() 将被调用来创建 request 对象。该方法仅仅会被 Scrapy 调用一次, 因此可以将其实现为生成器。该方法的默认实现是使用 start_urls 的 URL 生成 request
parse(response)	当 response 没有指定回调函数时, 该方法是 Scrapy 处理下载的 response 的默认方法。parse 负责处理 response 并返回处理的数据及(/或)跟进的 URL。Spider 对其他的 request 的回调函数也有相同的要求。该方法及其他的 request 回调函数必须返回一个包含 request 及(或)Item 的可迭代的对象
closed(reason)	当 Spider 关闭时, 该函数被调用。该方法提供了一个替代调用 signals.connect() 来监听 spider_closed 信号的快捷方式

### 4.1.2 通用蜘蛛

Scrapy 默认对特定爬取进行优化。这些站点一般被一个单独的 Scrapy Spider 处理, 不过这并不是必须的(例如, 也有通用的爬虫能处理任何给定的站点)。

除了这种爬取完某个站点或没有更多请求就停止的“专注的爬虫”, 还有一种通用的爬取类型, 其能爬取大量(甚至是无限)的网站, 仅仅受限于时间或其他的限制。这种爬虫叫作“通用爬虫(broad crawls)”, 一般用于搜索引擎。

通用爬虫一般有以下特性:

- 爬取大量(一般来说是无限)的网站而不是特定的一些网站。
- 不会将整个网站都爬取完毕, 因为这是不实际(或者说不可能完成)的。相反, 其



会限制爬取的时间和数量。

- 在逻辑上十分简单（相较于具有很多提取规则的复杂的 Spider），数据会在另外的阶段进行后处理（post-processed）；
- 并行爬取大量网站以避免被某个网站限制爬取的速度（每个站点爬取速度很慢但同时爬取很多站点）。

Scrapy 提供了 4 个通用的蜘蛛来处理一些普遍的爬取任务，它们分别是：

- XMLFeedSpider——用于爬取符合 XML 文档格式的蜘蛛基类；
- CSVFeedSpider——用于爬取 CSV 文件的蜘蛛；
- CrawlSpider——用于进行间接式递进爬取的蜘蛛；
- SitemapSpider——从 Sitemap.xml 文件跟随进入网站进行深度爬网的蜘蛛。

#### 4.1.2.1 XMLFeedSpider

XMLFeedSpider 被设计用于通过迭代各个节点来分析 XML 源（XML feed）。迭代器可以从 iternodes、XML 和 HTML 中选择。鉴于 XML 和 HTML 迭代器需要先读取所有 DOM 再分析性能问题，一般还是推荐使用 iternodes。不过使用 HTML 作为迭代器能有效地应对错误的 XML。

这是一个很有意思的蜘蛛，同时也是一个非常有用的基类。也就是说，它不能直接使用，而需要继承它实现其子类才能使用。XMLFeedSpider 继承自 Spider 类，但它的行为与 Spider 略有不同。

（1）必须声明 iterator（枚举器）属性，指明是针对文档的根节点还是针对指定节点进行搜寻。

（2）不需要重写 parse 方法，而是重写 parse\_node。XMLFeedSpider 已经实现了 parse 方法，而且会按照 itertag 属性中指定的标签名称筛选出节点的集合，然后逐一调用 parse\_node 方法并将该节点（选择器）作为处理上下文参数传入。

XMLFeedSpider 的可重写属性有：

- iterator——用于声明枚举器的类型。可选值为 iternodes（默认）、xml、html。
- itertag——用于指定筛选哪些 XML 标签。默认值为 item。
- namespaces——具有特殊命名空间的 XML 文档可以通过此元组属性指定。

以下是采用 XMLFeedSpider 改写中新网新闻供稿的直接型爬网示例的代码：

```
# coding=utf-8
from scrapy.spiders import XMLFeedSpider
from ..items import FeedItem
```



```

class ChinaNewsXmlFeedSpider(XMLFeedSpider):
    name = 'chinanews-xml'
    start_urls = { "http://www.chinanews.com/rss/scroll-news.xml" }

    # 这是 XMLFeedSpider 的两个默认属性，由于值相同，所以无须在子类中改写
    # iterator = 'iternodes'
    # itertag = 'item'

    def parse_node(self, response, node):
        item = FeedItem()
        item['title'] = node.xpath('title/text()').extract_first()
        item['link'] = node.xpath('link/text()').extract_first()
        item['desc'] = node.xpath('description/text()').extract_first()
        item['pub_date'] = node.xpath('pubDate/text()').extract_first()

        yield item

```

与第3章用选择器改写的中新网示例的 `parse` 方法对比，上述代码只是少了 `for` 循环，而且不再使用 `yield` 返回 `FeedItem` 的对象迭代器。`parse_node` 就是针对单个节点进行处理的，`XMLFeedSpider` 已经为我们在 `parse` 中写好了针对 `itertag` 属性所生成的 XPath，并在执行后将结果输入到 `parse_node` 中。这就是 `XMLFeedSpider` 的本质。

由于 HTML5 的大面积推广和应用，现在绝大多数主流的网站都会采用 HTML5 作为编写网页的标准。又因 HTML5 本身就严格遵从 XML 的规范，所以我们可以使用 `XMLFeedSpider` 爬取绝大多数的网页，这样可以在一定程度上降低蜘蛛分析逻辑的复杂性，同时也减少了一定的代码量。

#### 附：XMLFeedSpider类参考

属性说明如下表所示。

属 性	说 明
iterator	<p>用于确定使用哪个迭代器的 string。可选项有：</p> <ul style="list-style-type: none"> <li>* <code>iternodes</code>——一个高性能的基于正则表达式的迭代器</li> <li>* <code>html</code>——使用 Selector 的迭代器。需要注意的是，该迭代器使用 DOM 进行分析，其需要将所有 DOM 载入内存，当数据量大时会产生问题</li> <li>* <code>xml</code>——使用 Selector 的迭代器。需要注意的是，该迭代器使用 DOM 进行分析，其需要将所有 DOM 载入内存，当数据量大时会产生问题。默认值为 <code>iternodes</code></li> </ul>





续表

属 性	说 明
itertag	一个包含开始迭代的节点名的 string
namespaces	一个由 (prefix, url) 元组 (tuple) 所组成的 list 对象。它定义了在该文档中会被 Spider 处理的可用的 namespace。prefix 和 uri 会被自动调用 register_namespace() 生成 namespace。可以通过在 itertag 属性中制定节点的 namespace

例如, 指定爬取 sitemap 文件的命名空间:

```
class YourSpider(XMLFeedSpider):
    namespaces = [('n', 'http://www.sitemaps.org/schemas/sitemap/0.9')]
    itertag = 'n:url'
    # ...
```

可覆盖方法参考如下表所示。

方 法	说 明
adapt_response(response)	该方法在 Spider 分析 response 前被调用。可以在 response 被分析之前使用该函数来修改内容 (body)。该方法接收一个 response 并返回一个 response (可以相同也可以不同)
parse_node(response, selector)	当节点符合提供的标签名时 (itertag) 该方法被调用。接收到的 response 及相应的 Selector 作为参数传递给该方法。该方法返回一个 Item 对象或者 request 对象, 或者一个包含二者的可迭代对象 (iterable)
process_results(response, results)	当 Spider 返回结果 (Item 或 request) 时该方法被调用。设定该方法的目的是在结果返回给框架核心 (framework core) 之前做最后的处理, 例如, 设定 Item 的 ID。其接收一个结果的列表 (list of results) 及对应的 response。其结果必须返回一个结果的列表 (list of results), 包含 Item 或者 request 对象

#### 4.1.2.2 CSVFeedSpider

CSVFeedSpider 是一种定向性很强的蜘蛛, 顾名思义, 它是专门用于爬取 CSV 文件的。虽然 Python 的原生类库中就有针对 CSV 文件的读取器, 而且很容易使用, 但 CSV 本来就是一种通用的文本标准, 也是我们经常碰到的一种数据源。Scrapy 甚至进行了进一步的简化, 为实战开发带来了非常多的便利。

CSVFeedSpider 在爬虫系统中常见吗? 以 CSV 为数据交互格式在国内应用得不算多, 毕竟这种格式是一种很老旧的文件格式, 我的实战经历中遇到最多的是用 CSV 作为企业间没有固定



格式的非结构化数据交换，这些 CSV 多是从 Excel 中导出的。这会带来许多问题，例如，Excel 会自作聪明地将一个大整数以科学计数法导出到 CSV 文本中，使得整数变成了完全无法转换的字符串，还有就是如果不采用 Unicode 作为编码，则导出的 CSV 文件会充满中文乱码。这些内容在后面的章节会有更加细致的讲述。

由于我遇到的很多电商企业在需要进行数据交换和同步时，他们自有的 CRM 都是一些非常老旧或者说根本没有应用良好 RESULT-API 调用条件的，大多会将一些 Excel 文件或者 CSV 文件放到某个静态文件服务器上供我们下载，因此我当时的设计就是直接采用 CSVFeedSpider 作为数据同步的工具，从几个不同的商家网站那里直接爬取 CSV 文件，然后在管道内重新加工为需要的数据格式后保存。这种使用爬虫来作为一种数据同步的手段可以作为学成虫术后进行应用时的一种参考。

与 XMLFeedSpider 一样，Scrapy 已经将必要的循环在 parse 方法中实现了，在使用 CSVFeedSpider 时，只需要重写它的 parse\_row 即可。另外，除了重写 parse\_row 来提取数据，CSVFeedSpider 还提供了三个属性选项来对使用进行一些调节：

- delimiter——设置每行字段值间的分隔符，设置为 None 时使用，作为分隔符。
- quotechar——设置采用的引号，避免因引号出现的字符串引用异常。
- headers——在 CSV 文件中包含的用来提取字段的名称列表。

parse\_row 方法比 XMLFeedSpider 的 parse\_node 更容易使用，它传入两个参数：response 和 row。row 是一个字典类型参数，因此我们可以用字典的方式来使用它。

该方法接收一个 response 对象及一个以提供或检测出来的 header 为键的字典（代表每行）。在该 Spider 中，也可以覆盖 adapt\_response 及 process\_results 方法来进行预处理（pre-processing）及后处理（post-processing）。

```
from scrapy.spiders import CSVFeedSpider

class ProductCSVSpider(CSVFeedSpider):
    name = 'example.com'
    start_urls = ['http://www.example.com/feed.csv']
    headers = ['name', 'price']

    def parse_row(response, row):
        product = Product()
        product['name'] = row['name']
```





```
product['price'] = float(row[s'price'])
return product
```

**注：**与 XMLFeedSpider 的 parse\_node 一样，parse\_row 必须返回一个 Item 实例。

#### 4.1.2.3 示例：爬取全国空气质量的CSV数据

这是一个比较有代表性的示例，我在编写本书时偶然找到了一个仍然有大量 CSV 数据下载的网站。数据比较单一，只有全国 PM2.5 空气质量的检测数据。之所以说它比较有代表性，是因为：

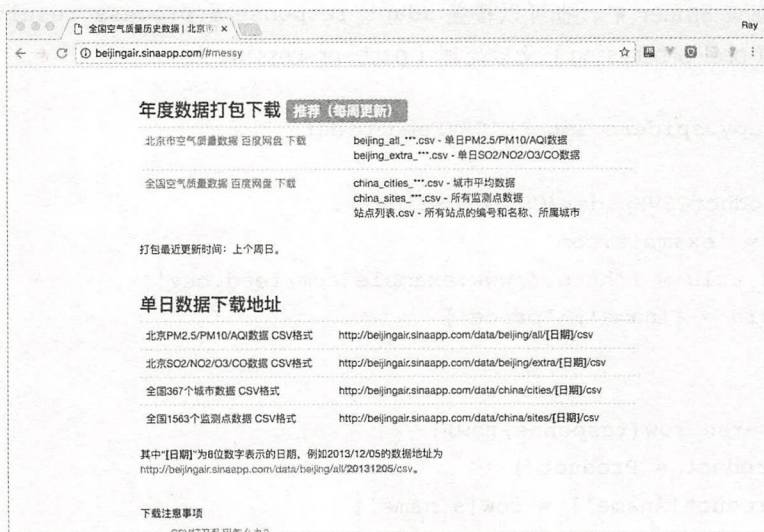
- 数据格式非常糟糕；
- 天气数据量非常大；
- 数据每天都持续更新。

这个示例的设计思路比较简单，具体如下：

- (1) 分析目标网站的爬取 URL 的生成规则（用于生成蜘蛛的 start\_urls）。
- (2) 分析 CSV 的数据结构及需要提取的具体数据范围。
- (3) 定义 CityItem 对象保存爬取的数据。
- (4) 编写蜘蛛。
- (5) 重整数据结构并保存于本地。

#### URL规则

下图所示就是这个网站所提供的数据 URL 的生成方式。



获取全国 367 个城市的数据的 URL 生成方式是：

```
http://beijingair.sinaapp.com/data/china/cities/[日期]/csv
```

这个网站的数据是从 2013 年开始提供的，如果要将 2013 年至今的数据一次性爬下来只需要按照“年年年年月月日日”的格式生成一个系列的 URL 序列作为 `start_urls` 就可以了。但我在此并不会这样写，主要基于几个原因：首先爬取的数据量会非常大，产生的数据请求也不少，没有必要因为要说明一个简单示例给爬取的网站带来巨大的网络负担，而且一旦对方有反爬机制，蜘蛛就会被直接封掉。其次，在本书后面章节才会讲述如何应对具有反爬机制网站，为了降低本示例代码的复杂性，此处只采用了固定的几个 URL 来进行爬取。

### 数据格式

根据上述的文件下载规则下载并打开一份 CSV 数据来了解目标数据的结构，由于这个 CSV 将城市信息作为列来存放，因此只截取了三个城市的前几条数据来展示，具体结构如下：

```
date,hour,type,北京,天津,石家庄, ...
20171101,0,AQI,137,108,139, ...
20171101,0,PM10,150,135,199 , ...
20171101,0,SO2,3,21,24, ...
```

- **date:** 检测日期；
- **hour:** 检测时间（小时）；
- **type:** 类型；
- 前三个类型以后的都是城市名称，对应的值是具体的指示值。

按照这个结构的逻辑，我们不难设计出以下数据项：

```
from scrapy import Item,Field

class City(Item):
    name = Field() # 城市名
    norm = Field() # 指标类型
    value = Field() # 值
    date = Field() # 检测日期
```





## 蜘蛛

这里有一个非常有参考性的内容，在之前的例子中，在 `parse` 中只会返回单条数据。

```
from ..items import City
from scrapy.spiders import CSVFeedSpider
from cities import names

class WeatherSpider(CSVFeedSpider):
    name = 'weather'
    start_urls = ['http://beijingair.sinaapp.com/data/china/cities/20171220/csv']

    def parse_row(response, row):
        city = City()
        for name in names:
            if row[name] != None:
                city['name'] = name
                city['norm'] = row['type']
                city['value'] = row[name]
                city['date'] = "%s-%s-%s" % (row['date'][0, 3], row['date'][4,
5], row['date'][6, 7])
        yield city
```

如果需要同时爬取多个 CSV，则可以将 `start_urls` 中的值从一个函数返回，例如：

```
start_urls = gen_urls()

def gen_urls:
    urls = []
    # 这里编写按时间范围生成 URLs 的逻辑
    return urls
```

最后，在命令行中运行以下指令，将爬取结果保存至本地：

```
$ scrapy crawl weather -o weather.json -f json
```



#### 4.1.2.4 CrawlSpider

CrawlSpider 可以说是一个非常实用的通用蜘蛛。在讲述它的用法之前，我们先来回顾一下前文中提到的一个重要内容：广度爬网与深度爬网。我们在设计爬网程序时经常会遇到这样的问题：很多情况下只知道某个目标网站上有我们需要的数据，却不清楚这些数据分散在网站的哪些地方。比如说有些数据放在首页，有些数据却分散在很深层次的链接内，如果每个网页都逐一地去人工分析，则会产生极为庞大的工作量，对于某些大型的门户网站，由于数据极其庞大，采用人工分析几乎是不可能的。如果说我们的蜘蛛认识路，知道目标网站上有哪些链接可以爬，链接里还嵌套有哪些链接，那么是否就能取代这种几乎不可能完成的人工任务呢？这其实是一个相当大的课题，从专业理论的角度来讲，这个做法叫作建立爬网索引。

爬网索引是搜索引擎中很常用的一种技术（CrawlSpider 这种蜘蛛在搜索引擎中也是很常见的），现在进行一次换位思考：假如我们建立了一个新的网站要进行推广，直接有效的办法应该就是登录搜索引擎。所谓的登录搜索引擎就是向搜索引擎提供我们网站的公网地址（URL）。一般来说，一周后我们网站上的网页的内容就能出现在搜索引擎上，其他用户就能通过相关的关键字找到我们网站上的特定内容了。那搜索引擎是怎么单凭一个网络地址就将网站的内容存到服务器上的呢？这当然归功于搜索引擎的蜘蛛了。在提交了网址后，搜索引擎就会到我们的网站根目录下找 robot.txt 文件以获取哪些内容可以爬、哪些内容不应该爬，这已经是一种约定俗成的做法了，也可以称为“文明爬网”。这一点在设计爬虫系统时也是需要注意的，网站上有些地方站长可能并不欢迎爬虫进入，甚至在那些网址中可能埋有网页地雷或者封禁爬虫的程序。

robot.txt 是针对爬虫的简单通用协议，它有 allow 和 deny 两种规则：

```
ALLOW http://www.example.com/news
DENY http://www.example.com/system/*
```

这是一种“规则”，搜索引擎会遵守这个规则，从首页开始将<a>标记内 href 所指向的 URL 提取出来，然后顺着这些 URL 继续深入下一层再进行链接地址的提取，如果<a>标记用 follow="nofollow" 指定，蜘蛛就会终止对该链接的深入，这是蜘蛛与网页间沟通的一种语言。当我们真的需要文明爬网时就遵守这样的规则，否则对方就有可能对我们的爬虫采取一些应对的手段了，毕竟“互联网不是战场”，还是要注意爬虫的素质。蜘蛛会将这一切的 URL 和 URL 上网页<head>标记中的一些必要信息存到搜索引擎的索引服务器中，并对<head>标记中提供的内容进行一次基本的排序以便于用户搜索，这就是搜索引擎建立搜索索引的简单原理。

CrawlSpider 就是用来实现这种多层次深度爬网的蜘蛛，它会按照我们所指定的一系列规则到目标网站上将相应的链接甚至链接的内容取下来。这是 Scrapy 团队设计 CrawlSpider 的一个非常棒的地方，如果 CrawlSpider 只能爬取链接地址，则在 CrawlSpider 完成任务后还得派出另一个蜘蛛按照 URL 索引重新造访目标网站一次。这样做会导致网络资源的浪费，而且





蜘蛛的频繁造访很可能会触发对方网站上的一些反爬网机制，导致爬取的失败。

### 使用CrawlSpider

CrawlSpider 是一个支持多层次深度爬网的通用爬虫，由于比较特殊，所以需要一些类与之配合。它们分别是：

- Rule——用于制定的爬网规则与回调方法；
- LinkExtractor——用于从 Response 对象中提取链接。

#### ➤ 链接提取器 (LinkExtractor)

链接提取器 (LinkExtractors) 是由 Scrapy 内置的用于从网页 (scrapy.http.Response 对象) 中抽取允许被跟随 (follow) 进入的链接对象。

follow 是<a>的一个属性，<a href="链接地址" follow="true">链接标题</a>一旦被标记上 follow，就表示这个链接的目标地址允许爬虫进入，follow 属性在<a>没有被指定时默认表示为真，如果要同时设置当前页面内的所有<a>标记的 follow 属性，还可以在<head>元素中使用<link>进行统一设置。

Scrapy 默认提供了好几种链接提取器，例如，HtmlParserLinkExtractor、LxmlLinkExtractor、SgmlLinkExtractor 和 RegexLinkExtractor。但在最新的官方版本中并不建议直接指明采用的是哪一种链接提取器，而改用 LinkExtractor 替代（明显是将过多的子类进行重新抽象的重构结果），本质上 LinkExtractor 就是 LxmlLinkExtractor。也就是说，使用时只要这样引入就可以了：

```
from scrapy.linkextractors import LinkExtractor
```

每个 LinkExtractor 有唯一的公共方法：extract\_links，它接收一个 response 对象，并返回一个 scrapy.link.Link 对象数组。

一般来说，链接提取器是配合 CrawlSpider 类一同使用的，由于 CrawlSpider 可以通过重写内部的爬取对象规则以指定链接的提取范围，在 CrawlSpider 使用 LinkExtractor 时无须重写 parse 方法。但也可以应用在 Spider 中，即使不是从 CrawlSpider 继承也可以。因为它的目的很简单：从响应对象中提取链接。

LinkExtractor 在 CrawlSpider 中的使用非常简单，只需要将其进行实例化并传入一个 Rule 对象的构造方法中即可，详细的使用方法会在下文中间介绍。



### ➤ 爬取规则 (Crawling Rule)

爬取规则也是一个非常简单的 Python 类，在 CrawlSpider 实例化后并不需要再进行其他手工的调用，CrawlSpider 会按照 Rule 所指定的规则执行爬取操作。

为了更好地理解 CrawlSpider、Rule 和 LinkExtractor 是如何协同工作的，直接进入我们已经很熟悉的新网供稿爬取示例，这次改写会更为复杂，要从供稿列表开始爬取，然后将每个栏目中的 RSS 都分别爬取一次。换句话说，这是一个同时实现直接爬取和间接爬取的综合型蜘蛛。

```
import scrapy
from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors import LinkExtractor

class ChinanewsCrawlSpider(CrawlSpider):
    name = 'chinanews.com'
    allowed_domains = ['www.chinanews.com']
    start_urls = ['http://www.chinanews.com/rss/rss_2.html']

    rules = (
        Rule(LinkExtractor(allow=('*\.xml', )), callback='parse_items'),
    )

    def parse_items(self, response):
        selector = Selector(response);
        for node in selector.xpath('//item').extract():
            item = FeedItem()
            item['title'] = node.xpath('title/text()').extract_first()
            item['link'] = node.xpath('link/text()').extract_first()
            item['desc'] = node.xpath('description/text()').extract_first()
            item['pub_date'] = node.xpath('pubDate/text()').extract_first()
            yield item
```

以上代码实际上完成了一次嵌套式的爬取动作，蜘蛛会先从 start\_urls 指定的 URL 开始爬取，当返回响应对象后，CrawlSpider 会在 parse 方法中根据 rules 定义的规则（这里只允许爬取后缀名为\*.xml 的文件）对 XML 地址重新发起请求，待这个请求返回响应时将响应对象 response 传入由 callback 参数指定的 parse\_items 函数中进行处理。

如果 Rule 的构造函数中没有指定 callback 函数，那么 CrawlSpider 就会根据当前的规





则继续对返回的 `response` 中的链接对象（这些链接对象是由 `LinkExtractor` 自动提取的）深入进去，直到返回的响应对象中已经没有符合规则匹配的连接为止。

不要尝试覆盖重写 `parse` 方法，否则 `CrawlSpider` 会失去它原有的功效。

这就是 `CrawlSpider`、`Rule` 和 `LinkExtractor` 的协同用法，为了能更清楚地了解 `Rule` 的细节，以下是它的构造函数的定义和参数说明：

```
class scrapy.contrib.spiders.Rule(link_extractor, callback=None,
cb_kwargs=None, follow=None, process_links=None, process_request=None)
```

- `link_extractor`——一个链接提取器对象，其定义了如何从爬取的页面提取链接。
- `callback`——一个 callable 或 string（该蜘蛛中同名的函数将会被调用）。从链接提取器中获取链接时会调用该函数。该回调函数接受一个 `response` 作为其第一个参数，并返回一个包含 `Item` 及（或）`request` 对象（或者两者的子类）的列表（list）。
- `cb_kwargs`——包含传递给回调函数的参数（keyword argument）的字典。
- `follow`——一个布尔（boolean）值，指定了根据该规则从 `response` 中提取的连接是否需要跟进。如果 `callback` 为 `None`，`follow` 的默认设置为 `True`，否则默认为 `False`。
- `process_links`——一个 callable 或 string（该 Spider 中同名的函数将被调用）。从 `link_extractor` 中获取链接列表时会调用该函数。该方法主要用来过滤。
- `process_request`——一个 callable 或 string（该 Spider 中同名的函数将被调用）。该规则提取每个 `request` 时都会调用该函数。该函数必须返回一个 `request` 或者 `None`（用来过滤 `request`）。

**注意：**在编写爬虫规则时，避免使用 `parse` 作为回调函数，由于 `CrawlSpider` 使用 `parse` 方法来实现其逻辑，如果覆盖了 `parse` 方法，则 `crawl spider` 会运行失败。

`rules` 包含一个（或多个）`Rule` 对象的集合（list）。每个 `Rule` 对爬取网站的动作定义了特定表现。如果多个 `Rule` 匹配了相同的链接，则根据它们在本属性中被定义的顺序，第一个会被使用。

该 Spider 也提供了一个可复写（overrideable）的方法：

```
parse_start_url(response)
```

当 `start_url` 的请求返回时，该方法被调用。该方法分析最初的返回值并且必须返回一个



Item 对象或者一个 Request 对象，又或者一个可迭代的包含二者的对象。

### ➤ CrawlSpider 的使用建议

CrawlSpider 是一匹有野性的烈马，如果你不好好去设计爬取规则，它可能就会在对方网站上横冲直撞，直接导致对方网站访问缓慢，如果请求频次过高还能被认为是一种“攻击”。如果对方网站上有反爬机制的话，一定会第一时间封禁你的爬虫 IP，导致爬网失败。

所以使用 CrawlSpider 时要牢记两点：

- 设计尽量精确的爬取规则，减少爬网数量。
- 降低蜘蛛的并发频率（本书下文会有说明）。

#### 4.1.2.5 示例：某代理网站爬虫

本节将展示如何继承 CrawlSpider，快速地从某代理网址上采集可用的代理服务器的蜘蛛。下图为某代理的网站。

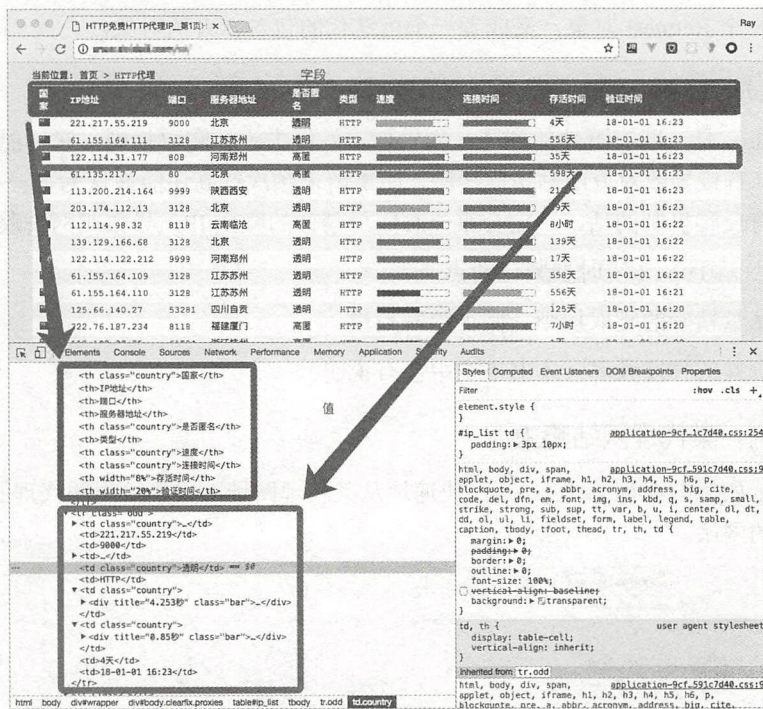
国家	IP地址	端口	服务器地址	域名	类型	速度	连接时间	存活时间	验证时间
中国	221.217.59.219	9000	北京	透明	HTTP		4天	18-01-01 16:23	
中国	61.155.164.111	3128	江苏苏州	透明	HTTP		556天	18-01-01 16:23	
中国	122.114.31.177	808	河南郑州	高匿	HTTP		36天	18-01-01 16:23	
中国	61.135.217.7	80	北京	高匿	HTTP		596天	18-01-01 16:23	
中国	113.200.214.164	9999	陕西西安	透明	HTTP		218天	18-01-01 16:23	
中国	203.174.112.13	3128	北京	透明	HTTP		49天	18-01-01 16:23	
中国	112.114.98.32	8118	云南临沧	高匿	HTTP		8小时	18-01-01 16:22	
中国	139.129.166.68	3128	北京	透明	HTTP		139天	18-01-01 16:22	
中国	122.114.122.212	9999	河南郑州	透明	HTTP		17天	18-01-01 16:22	
中国	61.155.164.109	3128	江苏苏州	透明	HTTP		556天	18-01-01 16:22	
中国	61.155.164.110	3128	江苏苏州	透明	HTTP		556天	18-01-01 16:21	
中国	125.66.140.27	53281	四川自贡	透明	HTTP		125天	18-01-01 16:20	
中国	222.76.187.234	8118	福建厦门	高匿	HTTP		7小时	18-01-01 16:20	
中国	115.193.99.35	61234	浙江杭州	高匿	HTTP		1天	18-01-01 16:20	
中国	59.38.62.250	9797	广东珠海	透明	HTTP		7小时	18-01-01 16:16	
中国	182.96.195.66	8118	江西南昌	高匿	HTTP		13小时	18-01-01 16:15	
中国	111.195.65.242	8123	北京	高匿	HTTP		1分钟	18-01-01 16:15	
中国	115.193.99.173	61234	浙江杭州	高匿	HTTP		20小时	18-01-01 16:14	
中国	115.239.42.232	3128	浙江绍兴	透明	HTTP		59分钟	18-01-01 16:14	
中国	125.46.0.62	53281	河南济源	透明	HTTP		138天	18-01-01 16:12	
中国	121.205.254.158	28764	福建莆田	高匿	HTTP		24天	18-01-01 16:11	
中国	114.212.80.2	3128	江苏南京	透明	HTTP		2天	18-01-01 16:10	
中国	180.114.150.95	808	江苏无锡	高匿	HTTP		19小时	18-01-01 16:10	
中国	1.196.161.162	9999	河南商丘	透明	HTTP		17小时	18-01-01 16:10	
中国	113.109.248.10	9797	广东广州	透明	HTTP		3天	18-01-01 16:10	
中国	61.155.164.107	3128	江苏苏州	透明	HTTP		556天	18-01-01 16:08	

相对来说这是一个非常容易被爬取的网站，其采用<table>元素来列举所有代理服务器的数据，因为<table>元素行列明确，因此很方便用代码来枚举其中的内容。

### 元素提取分析

首先，我们用 Chrome 的开发人员工具查看整列表的网页元素结构，如下图所示。





根据此结构我们可以先定义爬取的数据项，以确定采集的目标内容：

```
from scrapy import Item, Field

class ProxyItem(Item):
    ip = Field()           # 地址
    port = Field()         # 端口
    speed = Field()        # 连接速度
    connection_time = Field() # 连接时间
    ttl = Field()          # 存活时间
    protocol = Field()     # 连接协议
    validated = Field()    # 是否有效 (标志位)
```

由于此处有一个时间值是从“存活时间”中采集的，这是一个被格式化输出的文字内容，我们需要编写一个函数，将其内容进行重新格式化和再提取。

`_duration_to_millisecond` 函数就是将 5 秒、2 天等不可计算的字符串数据统一转换成以毫秒为单位的整数。

```
def _duration_to_millisecond(val):
    if val:
        if u'秒' in val:
            return int(float(val.replace(u'秒', '')) * 1000)
        if u'分钟' in val:
            return int(val.replace(u'分钟', '')) * 1000 * 60
        if u'小时' in val:
            return int(val.replace(u'小时', '')) * 1000 * 60 * 60
        if u'天' in val:
            return int(val.replace(u'天', '')) * 1000 * 60 * 60 * 24
    return 0
```

然后根据网页的结构可以看出网页元素 (XPath) 与 ProxyItem 实例之间的对应关系, 代码如下:

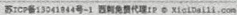
```
selector = Selector(response)
row_selectors = selector.xpath("//tr")

item['ip'] = row_selector.xpath("td[2]/text()").extract_first()
item['protocol'] = row_selector.xpath("td[6]/text()").extract_first().lower()
item['port'] = int(row_selector.xpath("td[3]/text()").extract_first())
item['connection_time'] = _duration_to_millisecond(connection_time_str)
item['speed'] = _duration_to_millisecond(row_selector.xpath('td[7]/div/
@title').extract_first())
item['ttl'] = _duration_to_millisecond(row_selector.xpath("td[9]/text()").
extract_first())
item['validated'] = row_selector.xpath("td[10]/text()").extract_first()
```

## 处理分页

通过以上的分析内容, 我们的蜘蛛已经完成了绝大部分, 接下来要考虑如何进行递进式的爬取了。将网页拉到最底可以见到一个分页器, 如下图所示。





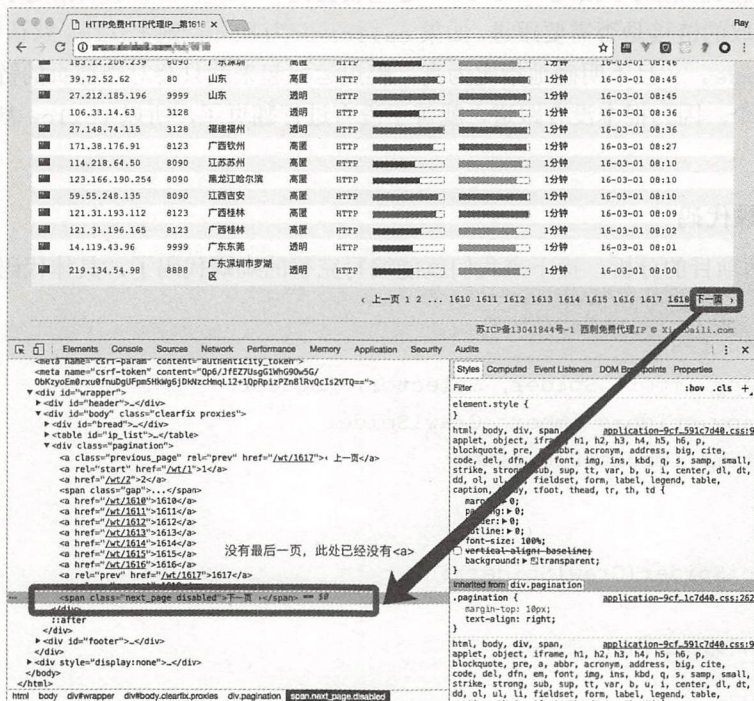
个链接是什么样的，如下图所示。

指向下一页

“下一页”就是链接标记，它指向的是一个“/wt/<页码>”的页面：

```
<a class="next_page" rel="next" href="/wt/3">下一页 </a>
```

然后直接跳到最后一页，看看“下一页”会变成什么，如下图所示。



当切换至最后一页时，“下一页”就已经不再是<a/>元素，而是一个<span>元素了，这样得到了基本分析思路。

CrawlSpider 的使用思路：

- (1) 找到递进深入的入口——“下一页”链接；
- (2) 分析递进链接的生成规则——“/xicidaili/wt/<正数页码>”。

这样就可以构造 Rules 了，具体代码如下所示。

```
start_urls = ['http://某代理网站网址/wt/1'] # 从第一页开始

rules = (
    Rule(LinkExtractor(allow=('/wt/*'), ), follow=True, callback='parse_items'),
)
```



由于数字链接页码可能会与“下一页”的链接重复,我们并不需要担心 Scrapy 会爬取重复性数据。Scrapy 早就为我们考虑到了这一点,它已经搭载并默认启动了一个去重过滤器(更多去重过滤器的内容会在高阶虫术的“去重处理”中详细介绍)。

而另一个需要注意的参数是 `follow`, 这里设置为 `True`, 这个参数指定了根据该规则从 `response` 中提取的链接是否需要跟进。如果 `callback` 为 `None`, 则 `follow` 默认设置为 `True`, 否则默认为 `False`。一旦不明确地指定为真, 爬虫运行起来就只会对当前页的链接爬取一次, 那么就只有 13 个。因为根本没有跟进, 就等于没有进行翻页了, 所以 `follow` 参数必须设置为 `True`。

### 完整的蜘蛛代码

经过以上两项目的分析, 接下来我们就能编写完整的蜘蛛代码了, 具体代码如下所示。

```
# coding=utf-8
from scrapy import Spider, Selector, Request
from scrapy.spiders import CrawlSpider
from ..items import ProxyItem

class XiciSpider(CrawlSpider):
    """
    某代理网站爬虫
    """
    name = "xici"
    allowed_domains = ["网址"]
    start_urls = ['http://网址/wt/1'] # 从第一页开始
    rules = (
        Rule(LinkExtractor(allow=('/wt/*')), follow=True, callback='parse_items'),
    )

    def parse_items(self, response):
        selector = Selector(response)
        row_selectors = selector.xpath("//tr")
        # items = []

        for row_selector in row_selectors[1:]:
            item = ProxyItem()
```

```

        connection_time_str = row_selector.xpath('td[8]/div/@title').
extract_first()
        item['ip'] = row_selector.xpath("td[2]/text()").extract_first()
        item['protocol'] = row_selector.xpath("td[6]/text()").extract_
first().lower()
        item['port'] = int(row_selector.xpath("td[3]/text()").extract_
first())
        item['connection_time'] = _duration_to_millisecond(connection_
time_str)
        item['speed'] = _duration_to_millisecond(row_selector.xpath
('td[7]/div/@title').extract_first())
        item['ttl'] = _duration_to_millisecond(row_selector.xpath
("td[9]/text()").extract_first())
        item['validated'] = row_selector.xpath("td[10]/text()").extract_
first()

        yield item

def _duration_to_millisecond(val):
    if val:
        if u'秒' in val:
            return int(float(val.replace(u'秒', '')) * 1000)
        if u'分钟' in val:
            return int(val.replace(u'分钟', '')) * 1000 * 60
        if u'小时' in val:
            return int(val.replace(u'小时', '')) * 1000 * 60 * 60
        if u'天' in val:
            return int(val.replace(u'天', '')) * 1000 * 60 * 60 * 24
    return 0

```

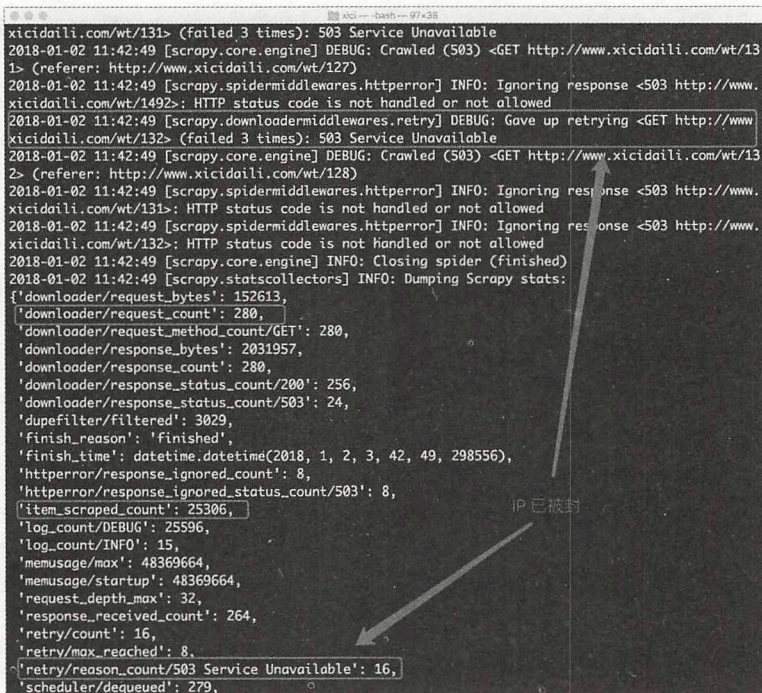
## 小结

完成以上代码后，就可以在命令行中执行以下指令开始采集该代理网站的数据：

```
$ scrapy crawl xici
```

从上文的截屏中得知有 1618 页，而每页又有上百个代理数据，简单计算一下应该能一次性爬取到 10 万条以上的数据量。只要运行以上代码，在很短的时间内会看到以下信息：





```

xiciidaili.com/wt/131> (failed 3 times): 503 Service Unavailable
2018-01-02 11:42:49 [scrapy.core.engine] DEBUG: Crawled (503) <GET http://www.xiciidaili.com/wt/131> (Referer: http://www.xiciidaili.com/wt/127)
2018-01-02 11:42:49 [scrapy.spidermiddlewares.httperror] INFO: Ignoring response <503 http://www.xiciidaili.com/wt/1492>: HTTP status code is not handled or not allowed
2018-01-02 11:42:49 [scrapy.downloadermiddlewares.retry] DEBUG: Gave up retrying <GET http://www.xiciidaili.com/wt/132> (failed 3 times): 503 Service Unavailable
2018-01-02 11:42:49 [scrapy.core.engine] DEBUG: Crawled (503) <GET http://www.xiciidaili.com/wt/132> (Referer: http://www.xiciidaili.com/wt/128)
2018-01-02 11:42:49 [scrapy.spidermiddlewares.httperror] INFO: Ignoring response <503 http://www.xiciidaili.com/wt/131>: HTTP status code is not handled or not allowed
2018-01-02 11:42:49 [scrapy.spidermiddlewares.httperror] INFO: Ignoring response <503 http://www.xiciidaili.com/wt/132>: HTTP status code is not handled or not allowed
2018-01-02 11:42:49 [scrapy.core.engine] INFO: Closing spider (finished)
2018-01-02 11:42:49 [scrapy.statscollectors] INFO: Dumping Scrapy stats:
{'download/request_bytes': 152613,
 'download/request_count': 280,
 'download/request_method_count/GET': 280,
 'download/response_bytes': 2031957,
 'download/response_count': 280,
 'download/response_status_count/200': 256,
 'download/response_status_count/503': 24,
 'dupefilter/filtered': 3029,
 'finish_reason': 'finished',
 'finish_time': datetime.datetime(2018, 1, 2, 3, 42, 49, 298556),
 'httperror/response_ignored_count': 8,
 'httperror/response_ignored_status_count/503': 8,
 'item_scraped_count': 25306,
 'log_count/DEBUG': 25596,
 'log_count/INFO': 15,
 'memusage/max': 48369664,
 'memusage/startup': 48369664,
 'request_depth_max': 32,
 'response_received_count': 264,
 'retry/count': 16,
 'retry/max_reached': 8,
 'retry/reason_count/503 Service Unavailable': 16,
 'scheduler/dequeued': 279,

```

Scrapy 不是运行完成，而是自动终止了，终止的原因是“503 Service Unavailable”（服务不可用），如果用浏览器再次打开这个网站，则会发现已经不能访问了，只会显示“Block”页面，现在只能很遗憾地告诉你 IP 被封了。我承认我在这个示例中埋下了一个大坑，那就是这个示例是绝对会被封 IP 的！因为在不做任何处理的情况下，Scrapy 会以 16 个并发请求来发出蜘蛛，这种采集速度是相当惊人的，从上图中就可以看出，仅仅在几十秒内我们就已经爬取了 25306 条数据！换个位置思考，如果你的网站在几十秒内被一个陌生的 IP 突然间发出了上百个请求，你会怎么处理呢？这其实就是遭遇了反爬系统的反击。

即使现在被封了 IP 也不要紧，换一个新的 IP 地址就可以了，这个坑不难爬出来。

之所以我要埋下这么一个坑，是想告诉你“爬虫虽强大，但反爬系统无处不在”。除了会与反爬系统正面遭遇，这个示例其实还存在其他坑，但也是我们掌握更高层次虫术的必经之路。现在需要我们停下来思考以下问题，带着这些问题在本书的后面章节中寻找答案。

(1) 如何才能不被反爬网机制发现？在不换 IP 的情况下能否持续地将所有数据一次性地收集完成？

(2) 代理服务器是有限制的，到了某个时间点会就失效，而且某些服务器的连接速度和访问速度也很低，根本没有收集的意义，那么对于这些“无用”的数据我们又将如何处理？

(3) 为了确保代理服务器数据的有效性，这会是一个持久的工作，那面对日益增长的数据

又应该如何处理呢?

#### 4.1.2.6 SitemapSpider

Sitemap 可方便网站管理员通知搜索引擎网站上有哪些可供抓取的网页。最简单的 Sitemap 形式就是 XML 文件,其列出网站中的网址及每个网址的其他元数据(上次更新的时间、更改的频率和相对于网站上其他网址的重要程度等),以便搜索引擎可以更加智能地抓取网站。简言之, Sitemap 就是为了蜘蛛能方便地爬取网站内容的一个入口式文件。

Google、雅虎和微软都支持一个被称为 XML 网站地图(xml Sitemaps)的协议,而百度 Sitemap 是指百度支持的收录标准,在原有协议上做出了扩展。百度 Sitemap 的作用是通过 Sitemap 告诉百度蜘蛛全面的站点链接,优化自己的网站。百度 Sitemap 分为三种格式:txt 文本格式、XML 格式、Sitemap 索引格式。

##### 格式

##### ► Google SiteMap

Google SiteMap Protocol 是 Google 自己推出的一种站点地图协议,此协议文件基于早期的 robots.txt 文件协议,并有所升级。在 Google 官方指南中指出加入了 Google SiteMap 文件的网站将更有利于 Google 网页爬行机器人的爬行索引,这样将提高索引网站内容的效率和准确度。文件协议应用了简单的 XML 格式,一共用到 6 个标签,其中关键标签包括链接地址、更新时间、更新频率和索引优先权。

```
<urlset xmlns="网页列表地址">
<url>
<loc>网址</loc>
<lastmod>2005-06-03T04:20-08:00</lastmod>
<changefreq>always</changefreq>
<priority>1.0</priority>
</url>
<url>
<loc>网址</loc>
<lastmod>2005-06-02T20:20:36Z</lastmod>
<changefreq>daily</changefreq>
<priority>0.8</priority>
</url>
</urlset>
```



### ➤ 百度 Sitemap

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset>
<url>
<loc>网页地址</loc>
<lastmod>2010-01-01</lastmod>
<changefreq>daily</changefreq>
<priority>1.0</priority>
</url>
</urlset>
```

### ➤ XML 标签

- changefreq: 页面内容更新频率;
- lastmod: 页面最后修改时间;
- loc: 页面永久链接地址;
- priority: 相对于其他页面的优先权;
- url: 相对于前 4 个标签的父标签;
- urlset: 相对于前 5 个标签的父标签。

### 适用性

对爬虫系统而言,如果能获得目标网站的 Sitemap,就等于得到了进入对方家门的钥匙,要拿里面的内容可以说是探囊取物般容易。Sitemap 本身又是以通用的标准格式来制作的,它就是蜘蛛爬网的路标。凡事有正反两面,采用 Sitemap 爬取方案也会遇到以下问题:

- Sitemap 可能只包含针对搜索引擎公开的一般性信息;
- Sitemap 可能更新不及时,导致包含的信息不全面;
- 很多网站在 robots.txt 中没有保存 Sitemap 对应的 URL;
- 为了“防虫”,当下大多网站都不公开 Sitemap 甚至没有 Sitemap。

### SitemapSpider

SitemapSpider 使得爬取网站时可以通过 Sitemaps 来发现爬取的 URL,其支持嵌套的 Sitemap,并能从 robots.txt 中获取 Sitemap 的 URL。其属性如下表所示。

属 性	说 明
sitemap_urls	包含要爬取的 URL 的 Sitemap 的 URL 列表 (list)，也可以指定为一个 robots.txt，Spider 会从中分析并提取 URL
sitemap_rules	一个包含 (regex, callback) 元组的列表 (list)
sitemap_follow	一个用于匹配要跟进的 Sitemap 的正则表达式的列表 (list)，其仅仅被应用在使用 Sitemap index files 来指向其他 Sitemap 文件的站点
sitemap_alternate_links	指定当一个 URL 有可选的链接时，是否跟进。有些非英文网站会在一个 URL 块内提供其他语言的网站链接，默认情况下所有的 Sitemap 都会被跟进

sitemap\_rules 附加说明：regex 是一个用于匹配从 Sitemap 提供的 URL 的正则表达式。regex 可以是一个字符串或者编译的正则对象 (compiled regex object)。callback 指定了匹配正则表达式的 URL 的处理函数。callback 可以是一个字符串 (Spider 中方法的名字) 或者是 callable。

例如：

```
sitemap_rules = [('/product/', 'parse_product')]
```

规则是按顺序进行匹配，之后第一个匹配才会被应用。如果忽略该属性，则 Sitemap 中发现的所有 URL 将被 parse 函数处理。

例如：

```
<url>
  <loc>http://example.com/</loc>
  <xhtml:link rel="alternate" hreflang="de" href="http://example.com/de"/>
</url>
```

当 sitemap\_alternate\_links 设置为开启时，两个 URL 都会被获取。当 sitemap\_alternate\_links 关闭时，只有 http://example.com/ 会被获取。

默认 sitemap\_alternate\_links 为关闭。

#### ➤ SitemapSpider 示例

因为很难在国内的网站上找到还在提供 Sitemap 的目标，所以本处的示例也只能是纸上谈兵，但反过来想，爬取国内网站基本不需要采用 SitemapSpider 了。简单的例子：使用 parse 处理通过 Sitemap 发现的所有 URL。



```
from scrapy.contrib.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ('http://example.webscraping.com/sitemap.xml')

    def parse(self, response):
        pass # ... scrape item here ...
```

用特定的函数处理某些 URL，其他的使用另外的 callback:

```
from scrapy.contrib.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ('http://example.webscraping.com/sitemap.xml')
    sitemap_rules = [
        ('/product/', 'parse_product'),
        ('/category/', 'parse_category'),
    ]

    def parse_product(self, response):
        pass # ... scrape product ...

    def parse_category(self, response):
        pass # ... scrape category ...
```

跟进 robots.txt 文件定义的 Sitemap 并只跟进包含 sitemap\_shop 的 URL:

```
from scrapy.contrib.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/', 'parse_shop'),
    ]
    sitemap_follow = ['/sitemap_shops']

    def parse_shop(self, response):
        pass # ... scrape shop here ...
```

在 SitemapSpider 中使用其他 URL:

```
from scrapy.contrib.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/', 'parse_shop'),
    ]

    other_urls = ['http://www.example.com/about']

    def start_requests(self):
        requests = list(super(MySpider, self).start_requests())
        requests += [scrapy.Request(x, self.parse_other) for x in self.other_urls]
        return requests

    def parse_shop(self, response):
        pass # ... scrape shop here ...

    def parse_other(self, response):
        pass # ... scrape other here ...
```

SitemapSpider 省去了很多跟进式爬网的代码,给蜘蛛的开发带来了很大的方便。与此同时,由于 Sitemap 文件的隐匿和网站防爬意识的提高, SitemapSpider 也变得鸡肋了。但是,我在开发爬虫系统时还是会习惯性地先看看目标网站上是否存有 Sitemap 文件。

### 4.1.3 蜘蛛中间件

Spider 中间件是介入 Scrapy 中的 Spider 处理机制的钩子框架,我们可以添加代码来处理发送给 Spiders 的 response 及 Spider 产生的 Item 和 request。

Scrapy 提供了两种中间件机制,一种是蜘蛛中间件,另一种则是下载器中间件。怎么来理解 Scrapy 提供的两种中间件机制,以及它们的作用与区别是什么呢?这要从 Scrapy 的整体结构与扩展机制来入手。首先 Scrapy 的扩展方式都是在配置文件 (settings.py/settings.cfg) 中将扩展的模块写入其中即可,配置就是所有模块的集成入口。而中间件则是一种特定的代码



扩展, 蜘蛛中间件就是 Scheduler 与蜘蛛之间的代码钩子, 允许我们在蜘蛛的处理周期内附加一些自定义的处理。通过中间件机制我们就能以增量式迭代法来扩充爬虫系统, 而不是在原有代码上不停地修改, 这样也极大提高了爬虫系统的模块化与可重用性。

通过蜘蛛中间件可以实现以下操作:

- 检测蜘蛛爬网的深度;
- 检测返回的 HTTP 响应是否有效;
- 在蜘蛛执行之前自动过滤无效的请求;
- 限制蜘蛛的某些行为;
- 其他用于控制蜘蛛行为的处理。

### 自定义 Spider 中间件

编写 Spider 中间件十分简单, 当我们使用 scrapy startproject 指令生成项目后, Scrapy 都会自动为我们创建一个中间件的模块 (放置在 middlewares.py 文件中), 打开这个文件就可以看到有一个蜘蛛中间件应该具有哪些方法接口:

```
from scrapy import signals

class MySpiderMiddleware(object):
    # 并不是所有的方法都必须被定义。如果以下方法没有被定义
    # 则 Scrapy 自动将对应的参数对象直接穿过中间件向下传递

    @classmethod
    def from_crawler(cls, crawler):
        # 这个方法是 Scrapy 用于构建蜘蛛的工厂方法
        s = cls()
        crawler.signals.connect(s.spider_opened, signal=signals.spider_opened)
        return s

    def process_spider_input(self, response, spider):
        # 每获得一个请求时就会调用此方法

        # 应该返回 None 或者引发一个异常
        return None
```

```

def process_spider_output(self, response, result, spider):
    # 当蜘蛛完成响应对象的处理就会调用该方法处理蜘蛛返回的结果

    # 必须返回一个可枚举的 request 对象、字典或 Item 对象
    for i in result:
        yield i

def process_spider_exception(self, response, exception, spider):
    # 当 process_spider_input() 或者蜘蛛引发异常后将调用本方法

    # 应该返回 None 或者一个可枚举的 response、dict 或者 Item 对象
    pass

def process_start_requests(self, start_requests, spider):
    # 当蜘蛛开始处理请求时本方法将被调用, 除了没有响应对象返回, 处理过程有点类似
    # process_spider_output() 方法

    # 必须返回请求对象 (不是 Items 对象)
    for r in start_requests:
        yield r

def spider_opened(self, spider):
    spider.logger.info('Spider opened: %s' % spider.name)

```

#### ➤ process\_spider\_input(response, spider)

当 response 通过 Spider 中间件时, 该方法被调用, 处理该 response。

process\_spider\_input() 应该返回 None 或者抛出一个异常。

- 如果其返回 None, 则 Scrapy 会继续处理该 response, 调用所有其他中间件直到 Spider 处理该 response。
- 如果其抛出一个异常 (exception), 则 Scrapy 不会调用任何其他中间件的 process\_spider\_input() 方法, 并调用 request 的 errback。errback 的输出会以另一个方向被重新输入中间件链中, 使用 process\_spider\_output() 方法来处理, 当其抛出异常时则调用 process\_spider\_exception()。

参数:

- response (response 对象) —— 被处理的 response;
- spider (Spider 对象) —— 该 response 对应的 Spider。



### ➤ `process_spider_output(response, result, spider)`

当 Spider 处理 `response` 返回 `result` 时, 该方法被调用。

`process_spider_output()` 必须返回包含 `request` 或 `Item` 对象的可迭代对象 (iterable)。

参数:

- `response` (`response` 对象) ——生成该输出的 `response`。
- `result` (包含 `request` 或 `Item` 对象的可迭代对象) ——Spider 返回的 `result`。
- `spider` (`Spider` 对象) ——其结果被处理的 `Spider`。

### ➤ `process_spider_exception(response, exception, spider)`

当 `Spider` 或(其他 `Spider` 中间件)的 `process_spider_input()` 抛出异常时, 该方法被调用。

`process_spider_exception()` 要么返回 `None`, 要么返回一个包含 `response` 或 `Item` 对象的可迭代对象 (iterable)。

- 如果其返回 `None`, 则 `Scrapy` 继续处理该异常, 调用中间件链中的其他中间件的 `process_spider_exception()` 方法, 直到所有中间件都被调用, 该异常到达引擎(异常将被记录并被忽略)。
- 如果其返回一个可迭代对象, 则中间件链的 `process_spider_output()` 方法被调用, 其他的 `process_spider_exception()` 将不会被调用。

参数:

- `response` (`response` 对象) ——异常被抛出时被处理的 `response`。
- `exception` (`Exception` 对象) ——被抛出的异常。
- `spider` (`Spider` 对象) ——抛出该异常的 `Spider`。

### ➤ `process_start_requests(start_requests, spider)`

该方法以 `Spider` 启动的 `request` 为参数被调用, 执行的过程类似于 `process_spider_output()`, 只不过其没有相关联的 `response`, 并且必须返回 `request` (不是 `Item`)。

其接收一个可迭代的对象 (`start_requests` 参数) 且必须返回另一个包含 `request` 对象的可迭代对象。

**注:** 当在 `Spider` 中间件实现该方法时, 必须返回一个可迭代对象 (类似于参数 `start_requests`) 且不要遍历所有的 `start_requests`。该迭代器会很大 (甚至是无限的), 进而导致内存溢出。`Scrapy` 引擎在其具有能力处理 `start request` 时将会拉起 `request`,

因此 start request 迭代器会变得无限大，由其他参数来停止 Spider（例如，时间限制或者 item/page 记数）。

参数：

- start\_requests（包含 request 的可迭代对象）——start requests。
- spider（Spider 对象）——start requests 所属的 Spider。

### 激活 Spider 中间件

要启用 Spider 中间件，可以将其加入 SPIDER\_MIDDLEWARES 设置中。该设置是一个字典，键为中间件的路径，值为中间件的顺序（order）。

具体做法如下所示。

```
SPIDER_MIDDLEWARES = {
    'myproject.middlewares.MySpiderMiddleware': 543,
}
```

SPIDER\_MIDDLEWARES 设置会与 Scrapy 定义的 SPIDER\_MIDDLEWARES\_BASE 设置合并（但不是覆盖），而后根据顺序（order）进行排序，最后得到启用中间件的有序列表：第一个中间件是最靠近引擎的，最后一个中间件是最靠近 Spider 的。

关于如何分配中间件的顺序请查看 SPIDER\_MIDDLEWARES\_BASE 设置，而后根据想要放置中间件的位置选择一个值。由于每个中间件执行不同的动作，我们的中间件可能会依赖于之前（或者之后）执行的中间件，因此顺序是很重要的。

如果想禁止内置的（在 SPIDER\_MIDDLEWARES\_BASE 中设置并默认启用）中间件，则必须在项目的 SPIDER\_MIDDLEWARES 设置中定义该中间件，并将其值赋为 None。例如，关闭 off-site 中间件：

```
SPIDER_MIDDLEWARES = {
    'myproject.middlewares.CustomSpiderMiddleware': 543,
    'scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware': None,
}
```

### 附：选择器对象参考

```
class scrapy.selector.Selector(response=None, text=None, type=None)
```

Selector 的实例是对选择某些内容响应的封装。



`response` 是 `HtmlResponse` 或 `XmlResponse` 的一个对象, 用来选择和提取数据。

`text` 是在 `response` 不可用时的一个 Unicode 字符串或 UTF-8 编码的文字。将 `text` 和 `response` 一起使用是未定义行为。

`type` 定义了选择器类型, 可以是 “html”、“xml” 或 `None` (默认)。

如果 `type` 是 `None`, 则选择器会根据 `response` 类型自动选择最佳的类型, 或者在和 `text` 一起使用时, 默认为 “html”。

如果 `type` 是 `None`, 并传递了一个 `response`, 则选择器类型将从 `response` 类型中推导, 如下:

- “html”——`HtmlResponse` 类型;
- “xml”——`XmlResponse` 类型;
- “html”——任意类型。

在其他情况下, 如果设定了 `type`, 则选择器类型将被强制设定, 而不进行检测。

```
xpath(query)
```

寻找可以匹配 `xpath query` 的节点, 并返回 `SelectorList` 的一个实例结果, 单一化其所有元素。列表元素也实现了 `Selector` 的接口。

`query` 是包含 `xpath` 查询请求的字符串。

**注解:** 为了方便起见, 该方法也可以通过 `response.xpath()` 调用。

```
css(query)
```

应用给定的 CSS 选择器, 返回 `SelectorList` 的一个实例。

`query` 是一个包含 CSS 选择器的字符串。

在后台, 通过 `cssselect` 库和运行 `.xpath()` 方法, CSS 查询会被转换为 XPath 查询。

**注解:** 为了方便起见, 该方法也可以通过 `response.css()` 调用。

```
extract()
```

串行化并将匹配到的节点返回一个 Unicode 字符串列表。结尾是编码内容的百分比。

```
re(regex)
```

应用给定的 `regex`, 并返回匹配到的 Unicode 字符串列表。

`regex` 可以是一个已编译的正则表达式，也可以是一个将被 `re.compile(regex)` 编译为正则表达式的字符串。

```
register_namespace(prefix, uri)
```

注册给定的命名空间，其将在 `Selector` 中使用。不注册命名空间，将无法从非标准命名空间中选择或提取数据。

```
remove_namespaces()
```

移除所有的命名空间，允许使用少量的命名空间 `xpaths` 遍历文档。

```
__nonzero__()
```

如果选择了任意的真实文档，则将返回 `True`，否则返回 `False`。也就是说，`Selector` 的布尔值是通过它选择的内容来确定的。

#### ➤ `SelectorList` 对象

```
class scrapy.selector.SelectorList
```

`SelectorList` 类是内建 `list` 类的子类，提供了一些额外的方法。

```
xpath(query)
```

对列表中的每个元素调用 `.xpath()` 方法，返回结果为另一个单一化的 `SelectorList`。

`query` 和 `Selector.xpath()` 中的参数相同。

```
css(query)
```

对列表中的各个元素调用 `.css()` 方法，返回结果为另一个单一化的 `SelectorList`。

`query` 和 `Selector.css()` 中的参数相同。

```
extract()
```

对列表中的各个元素调用 `.extract()` 方法，返回结果为单一化的 Unicode 字符串列表。

```
re()
```



对列表中的各个元素调用`.re()`方法, 返回结果为单一化的 Unicode 字符串列表。

```
__nonzero__()
```

列表非空则返回 `True`, 否则返回 `False`。

### ➤ 在HTML响应上的选择器样例

下面是一些 Selector 的样例, 用来说明一些概念。在所有的例子中, 我们假设已经有一个通过 `HtmlResponse` 对象实例化的 Selector, 如下:

```
sel = Selector(html_response)
```

从 HTML 响应主体中提取所有的`<h1>`元素, 返回`class:Selector`对象(即 `SelectorList` 的一个对象)的列表:

```
sel.xpath("//h1")
```

从 HTML 响应主体上提取所有`<h1>`元素的文字, 返回一个 Unicode 字符串的列表:

```
sel.xpath("//h1").extract()          # this includes the h1 tag
sel.xpath("//h1/text()").extract()   # this excludes the h1 tag
```

在所有`<p>`标签上迭代, 打印它们的类属性:

```
for node in sel.xpath("//p"):
    print node.xpath("@class").extract()
```

### ➤ 在XML响应上的选择器样例

下面是一些样例, 用来说明一些概念。在两个例子中, 我们假设已经有一个通过 `XmlResponse` 对象实例化的 Selector, 如下:

```
sel = Selector(xml_response)
```

从 XML 响应主体中选择所有的`<product>`元素, 返回 `Selector` 对象(即 `SelectorList` 对象)的列表:

```
sel.xpath("//product")
```





## 附: Scrapy内置Spider中间件参考

关于默认启用的中间件列表（及其顺序）请参考 `SPIDER_MIDDLEWARES_BASE` 设置。

### ➤ DepthMiddleware

```
class scrapy.contrib.spidermiddleware.depth.DepthMiddleware
```

`DepthMiddleware` 是一个用于追踪每个 `request` 在被爬取的网站中的深度的中间件, 其可以用来限制爬取的最大深度或类似的事情。

`DepthMiddleware` 可以通过下列设置进行配置:

- `DEPTH_LIMIT`——爬取所允许的最大深度, 如果为 0, 则没有限制。
- `DEPTH_STATS`——是否收集爬取状态。
- `DEPTH_PRIORITY`——是否根据其深度对 `request` 安排优先级。

### ➤ HttpErrorMiddleware

```
class scrapy.contrib.spidermiddleware.httperror.HttpErrorMiddleware
```

过滤出所有失败（错误）的 HTTP response, 因此 Spider 不需要处理这些 request。处理这些 request 意味着消耗更多资源, 并且使得 Spider 逻辑更为复杂。

根据 HTTP 标准, 返回值为 200~300 之间的为成功的 response。

如果想处理在这个范围之外的 response, 则可以通过 Spider 的 `handle_httpstatus_list` 属性或 `HTTPERROR_ALLOWED_CODES` 设置来指定 Spider 能处理的 response 返回值。

例如, 想要处理返回值为 404 的 response 则可以这么做:

```
class MySpider(CrawlSpider):
    handle_httpstatus_list = [404]
```

`Request.meta` 中的 `handle_httpstatus_list` 键也可以用来指定每个 request 所允许的 response code。

不过请记住, 除非知道在做什么, 否则处理非 200 返回一般来说是个糟糕的决定。

`settings.py` 的配置项: `HTTPERROR_ALLOWED_CODES`, 默认为 []。

忽略该列表中所有非 200 状态码的 response。

`HTTPERROR_ALLOW_ALL`, 默认: `False`。

忽略所有 response, 不管其状态值。

### ➤ OffsiteMiddleware

```
class scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware
```

过滤出所有 URL 不由该 Spider 负责的 request。

该中间件过滤出所有主机名不在 Spider 属性 `allowed_domains` 中的 request。

当 Spider 返回一个主机名不属于该 Spider 的 request 时，该中间件会做一个类似于下面的记录：

```
DEBUG: Filtered offsite request to 'www.othersite.com': <GET
http://www.othersite.com/some/page.html>
```

为了避免记录太多无用信息，其会对每个新发现的网站记录一次。因此，如果过滤出另一个 `www.othersite.com` 请求，则不会有新的记录。如果过滤出 `someothersite.com` 请求，则仍然会有记录信息（仅针对第一次）。

如果 Spider 没有定义 `allowed_domains` 属性，或该属性为空，则 `offsite` 中间件将会允许所有 request。

如果 request 设置了 `dont_filter` 属性，即使该 request 的网站不在允许列表里，`offsite` 中间件也会允许该 request。

### ➤ RefererMiddleware

```
class scrapy.contrib.spidermiddleware.referer.RefererMiddleware
```

根据生成 request 的 response 的 URL 来设置 request referer 字段。

settings.py 的配置项：REFERER\_ENABLED。

默认：True。

是否启用 referer 中间件。

### ➤ UrlLengthMiddleware

```
class scrapy.contrib.spidermiddleware.urllength.UrlLengthMiddleware
```

过滤出 URL 长度比 `URLLENGTH_LIMIT` 设置的 request 对象。

UrlLengthMiddleware 中间件只有一个配置项可以配置（更多内容请参考配置文档）：

`URLLENGTH_LIMIT`——允许爬取 URL 最长的长度。



## 4.2 爬虫系统的测试与调试

我发现很多年轻的程序员都不喜欢讨论测试和调试的技巧,甚至我曾见过某些“大神”写完程序连测都不测就直接完工的。可能很多人都认为测试是多余的,直接跳过测试可以“缩短”开发时间。其实不然,没有调试过或测试过的程序又以什么标准评判它已经完成了呢?即使程序在开发期间可以运行,但又如何确保部署后不会发生问题,又如何得知它发生过何种问题呢?相信这样的问题非常值得我们每个程序员思考。

爬虫系统是纯后端的,运行期几乎是没有任何运行界面的。我们所开发出来的产品是在视野之外运行的,在它运行结束后有可能引发大量的错误而我们却不得知。而当遇到采集数据量巨大、耗时的场景时,Bug 几乎都是致命性的。

在很多情况下,我们需要知道:

- (1) 爬虫是否成功获取响应结果。
- (2) 从响应结果中提取的数据是否正确。
- (3) 爬虫中某些变量的取值是否正确。
- (4) 代码逻辑是否被正确执行。
- (5) 在运行过程中是否曾出现异常,异常的跟踪情况又是如何的。
- (6) 部署后爬虫系统如果发生错误应该可以直接通知我。

以上这些都需要我们运用一些调试和测试的技巧才能得知,以使系统趋于稳定。调试和测试对于一个完整和稳定的系统来说是不可或缺的重要部分。

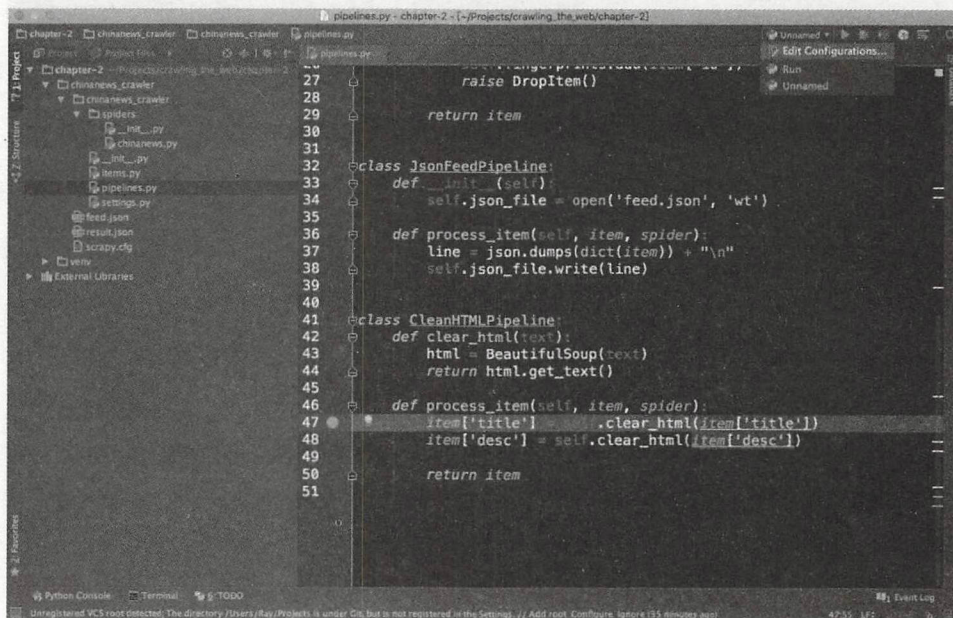
一般来说,在开发期的调试与测试都需要进行人工的观察,而当系统部署后就需要采用自动化的程序介入源代码中,并且设置一些监控点,一旦出现问题能及时执行某些恢复性或者通知性的操作。

### 4.2.1 开发期调试

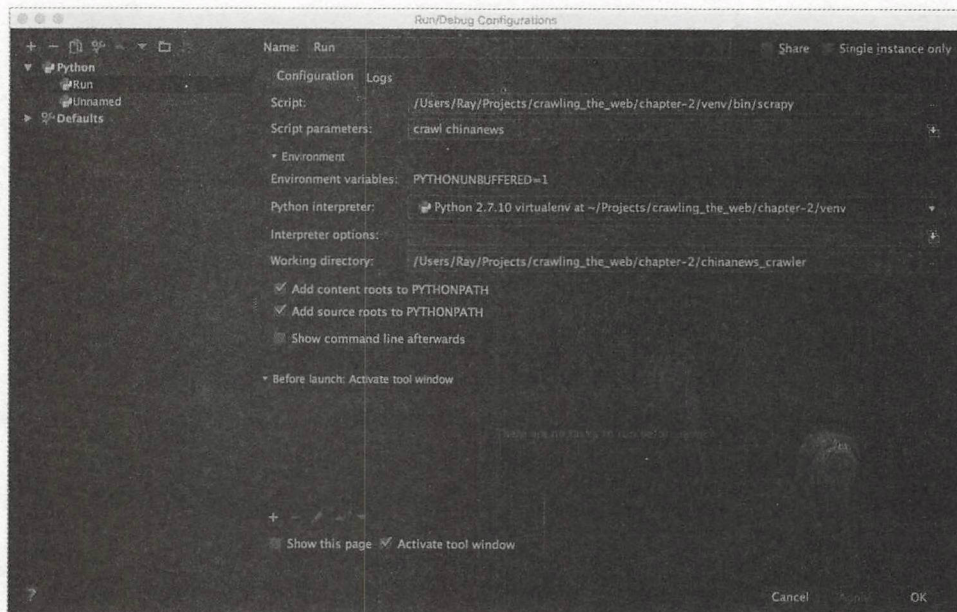
我做 Python 开发主要使用 PyCharm 的集成 IDE 作为主力开发工具,偶尔也会使用 Sublime 作为辅助开发工具。推荐使用 PyCharm 的原因是它集成了很多开发时必需的工具,例如,Git、代码高亮、自动代码格式化、语法检查等。吸引我一直使用它的原因是它的代码调试器非常好用,而且配置也非常简单,使用任何的 Python 框架都可以在 PyCharm 中得到调试器的支持。

调试代码时迅速找到代码中的问题或者 Bug 的唯一技巧就是使用断点,对于心存怀疑的代码先设置好断点,然后单步一步步地执行,一边执行一边观察相关的变量和实例化对象,这样做可以在极短的时间内找到问题所在。

在 PyCharm 中调试 Scrapy 项目非常容易，首先配置一个调试器的运行入口，如下图所示。

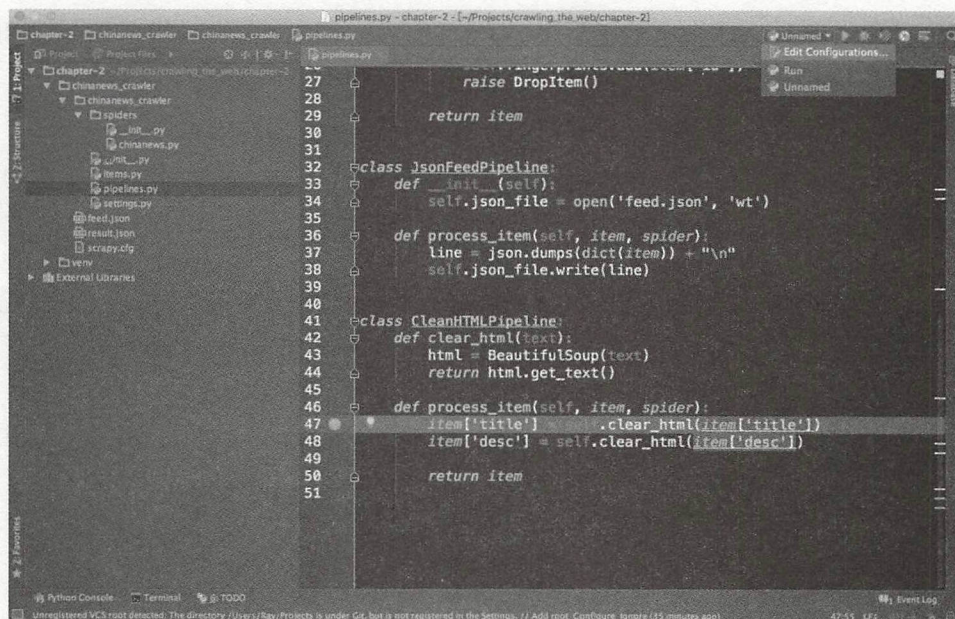


指定调试器的运行参数，具体如下图所示。

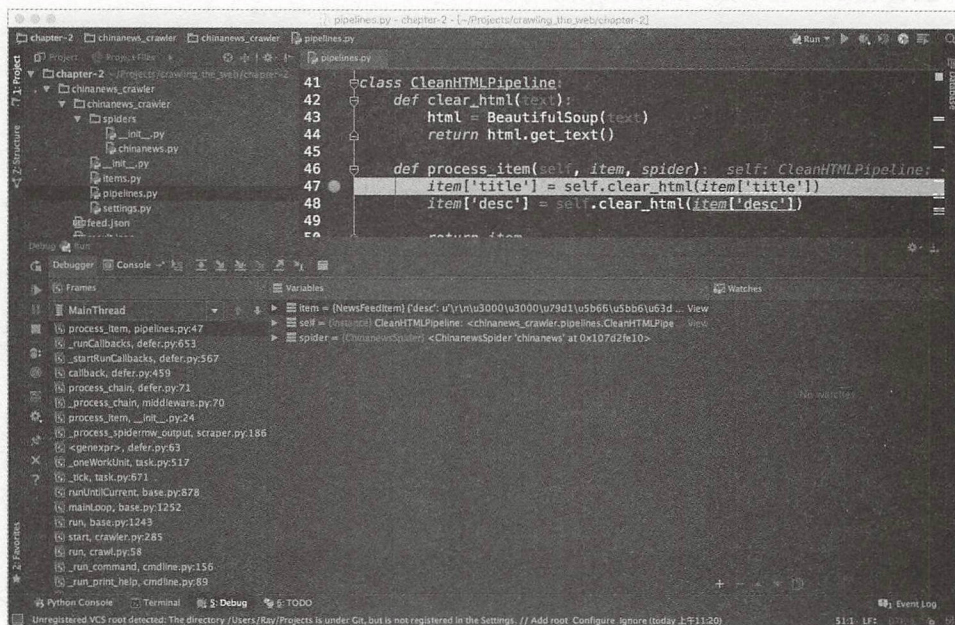


在需要中断程序的地方设置断点，如下图所示。





在 PyCharm 中以调试模式运行，当程序运行到断点处将自动中断运行，这样就可以一步步观察具体变量或者对象运行期的真实值了，如下图所示。



## 4.2.2 蜘蛛的测试

测试蜘蛛是一件挺烦人的事情，尤其是只能编写单元测试（Unit Test）时就烦人了。Scrapy 通过合同（contract）的方式来提供测试 Spider 的集成方法。

可以编写一个样例 url，设置多个条件来测试回调函数处理 response 的结果及测试 Spider 的回调函数。每个 contract 包含在文档字符串（docstring）中，以@开头。查看下面的例子：

```
def parse(self, response):
    """ This function parses a sample response. Some contracts are mingled
    with this docstring.

    @url http://www.amazon.com/s?field-keywords=selfish+gene
    @returns items 1 16
    @returns requests 0 0
    @scrapes Title Author Year Price
    """
```

该回调函数使用了三个内置的 contract 类来进行测试：

```
scrapy.contracts.default.UrlContract
```

该 contract (@url) 设置了用于检查 Spider 的其他 contract 状态的样例 URL。该 contract 是必需的，所有缺失该 contract 的回调函数在测试时会被忽略：

```
@url url
scrapy.contracts.default>ReturnsContract
```

该 contract (@returns) 设置 Spider 返回的 items 和 requests 的上界和下界。上界是可选的：

```
@returns item(s) | request(s) [min [max]]
scrapy.contracts.default.ScrapesContract
```

该 contract (@scrapes) 检查回调函数返回的所有 Item 是否有特定的字段：

```
@scrapes field_1 field_2 ...
```

使用 check 命令运行 contract 检查。



## 自定义 Contracts

如果想要比内置 Scrapy contract 更强大的功能,则可以在项目中创建并设置自己的 contract,并在 settings.py 文件中使用 SPIDER\_CONTRACTS 设置来进行加载:

```
SPIDER_CONTRACTS = {
    'myproject.contracts.ResponseCheck': 10,
    'myproject.contracts.ItemValidate': 10,
}
```

每个 contract 必须继承 scrapy.contracts.Contract 并覆盖下列三个方法:

```
class scrapy.contracts.Contract(method, *args)
```

参数:

- method (function)——contract 关联的回调函数;
- args (list)——传入 docstring 的 (以空格区分的) argument 列表 (list);
- adjust\_request\_args(args)——接收一个字典 (dict) 作为参数,该参数包含了所有 request 对象参数的默认值,该方法必须返回相同或修改过的字典;
- pre\_process(response)——该函数在 sample request 接收到 response 后,传送给回调函数前被调用,运行测试;
- post\_process(output)——该函数处理回调函数的输出。迭代器 (Iterators) 在传输给该函数前会被列表化 (listified)。

该样例 contract 在 response 接收时检查是否有自定义 header。在失败时发起一个 scrapy.exceptions.ContractFaild 异常来展现错误:

```
from scrapy.contracts import Contract
from scrapy.exceptions import ContractFail

class HasHeaderContract(Contract):
    """ Demo contract which checks the presence of a custom header
        @has_header X-CustomHeader
    """
    name = 'has_header'

    def pre_process(self, response):
```

```

for header in self.args:
    if header not in response.headers:
        raise ContractFail('X-CustomHeader not present')

```

### 运行contract

执行 contract 可以使用命令行工具 check，具体指令如下所示。

```
$ scrapy check -l
```

contract 是 Scrapy 在 0.15 版本之后加入的功能，虽然功能非常简陋，而且样本数据与测试期望只能被写死，但了胜于无，有测试手段总比没有测试手段要好。

## 4.2.3 蜘蛛的运行期调试

我们会经常遇到这样一种情况：程序员用很短的时间写完了代码且通过了本地测试，认为可以部署了。然而当代码真正部署到真实环境之中，总会出现这样或那样的问题。当然，如果你会配置 CI（持续集成）环境或者使用了 Docker 或 Vagrant 等虚拟环境，是可以完全将开发环境与部署环境做到一致化的，从而将上述情况的出现率降低甚至消灭。然而，如果不懂 CI、Docker 和 Vagrant 呢？当我们将代码部署到服务器上，又没有任何开发工具，如何进行远程的实地调试来稳定爬虫系统呢？

首先，要有一个蜘蛛的例子，考虑下面的蜘蛛代码：

```

import scrapy
from myproject.items import MyItem

class MySpider(scrapy.Spider):
    name = 'myspider'
    start_urls = (
        'http://example.com/page1',
        'http://example.com/page2',
    )

    def parse(self, response):
        # collect `item_urls`
        for item_url in item_urls:
            yield scrapy.Request(item_url, self.parse_item)

```



```

def parse_item(self, response):
    item = MyItem()
    # populate `item` fields
    # and extract item_details_url
    yield scrapy.Request(item_details_url, self.parse_details, meta=
{'item': item})

def parse_details(self, response):
    item = response.meta['item']
    # populate more `item` fields
    return item

```

简单地说, 该蜘蛛分析了两个包含 `Item` 的页面 (`start_urls`)。 `Item` 有详情页面, 所以我们使用 `request` 的 `meta` 功能来传递已经部分获取的 `Item`。

### parse命令

检查蜘蛛输出的基本方法是使用 `parse` 命令。这能让你在函数层 (`method level`) 上检查 `Spider` 各个部分的效果。 `parse` 命令十分灵活且易用, 不过不能在代码中调试, 只能纯粹观察输出结果。

查看特定 URL 爬取的 `Item`:

```

$ scrapy parse --spider=mypider -c parse_item -d 2 <item_url>
[ ... scrapy log lines crawling example.com spider ... ]

```

```
>>> STATUS DEPTH LEVEL 2 <<<
```

```
# Scraped Items -----
[{'url': <item_url>}]
```

```
# Requests -----
[]
```

使用 `--verbose` 或 `-v` 选项查看各个层次的状态:

```

$ scrapy parse --spider=mypider -c parse_item -d 2 -v <item_url>
[ ... scrapy log lines crawling example.com spider ... ]

```

```
>>> DEPTH LEVEL: 1 <<<
```

```
# Scraped Items -----
[]
```

```
# Requests -----
[<GET item_details_url>]
```

```
>>> DEPTH LEVEL: 2 <<<
```

```
# Scraped Items -----
[{'url': <item_url>}]
```

```
# Requests -----
[]
```

检查从单个 `start_url` 爬取的 Item 也是很简单的:

```
$ scrapy parse --spider=myspider -d 3 'http://example.com/page1'
```

## 在浏览器中打开

有时候想查看某个 `response` 在浏览器中显示的效果, 可以使用 `open_in_browser` 功能。下面是使用的例子:

```
from scrapy.utils.response import open_in_browser
```

```
def parse_details(self, response):
```

```
    if "item name" not in response.body:
```

```
        open_in_browser(response)
```

`open_in_browser` 会使用 Scrapy 获取的 `response` 来打开浏览器, 并且调整 `base tag`, 使得图片及样式 (`style`) 能正常显示。

## 日志

记录 (logging) 是另一个获取到蜘蛛运行信息的方法。虽然不是很方便, 但好处是 `log` 的内容在以后的运行中也可以看到:

```
from scrapy import log
```





```
def parse_details(self, response):
    item = response.meta.get('item', None)
    if item:
        # populate more `item` fields
        return item
    else:
        self.log('No item received for %s' % response.url,
                 level=log.WARNING)
```

Scrapy 提供了 5 层 logging 级别:

- CRITICAL——严重错误 (critical);
- ERROR——一般错误 (regular errors);
- WARNING——警告信息 (warning messages);
- INFO——一般信息 (informational messages);
- DEBUG——调试信息 (debugging messages)。

当爬虫是持久性运行的爬虫时, 由于数据量庞大, 因此在没有必要的情况下, 不建议记录 WARNING 级别以下的信息, 否则日志会成为吞噬服务器宝贵硬盘资源的怪兽。

### Scrapy 终端 (shell)

尽管 parse 命令对检查 Spider 的效果十分有用, 但除了显示收到的 response 及输出, 其对检查回调函数内部的过程并没有提供什么便利。如何调试 parse\_detail 没有收到 Item 的情况呢?

Scrapy 终端是一个交互终端, 供用户在未启动 Spider 的情况下尝试及调试爬取代码。其本意是用来测试提取数据的代码, 不过可以将其作为正常的 Python 终端, 在上面测试任何的 Python 代码。

该终端用来测试 XPath 或 CSS 表达式, 查看它们的工作方式及从爬取的网页中提取的数据。在编写 Spider 时, 该终端提供了交互性测试表达式代码的功能, 免去了每次修改后运行 Spider 的麻烦。

一旦熟悉了 Scrapy 终端, 会发现其在开发和调试 Spider 时发挥了巨大的作用。

#### ➤ 启动终端

可以使用 shell 来启动 Scrapy 终端:

```
$ scrapy shell <url>
```

<url>是要爬取的网页的地址。



### ➤ 使用终端

Scrapy 终端仅仅是一个普通的 Python 终端（或 IPython），其提供了一些额外的快捷方式。可用的快捷命令（shortcut）：

- `shelp()`——打印可用对象及快捷命令的帮助列表；
- `fetch(request_or_url)`——根据给定的请求（request）或 URL 获取一个新的 response，并更新相关的对象；
- `view(response)`——在本机的浏览器打开给定的 response，其会在 response 的 body 中添加一个<base>的标记，使得外部链接（例如，图片及 CSS）能正确显示。

注意，该操作会在本地创建一个临时文件，且该文件不会被自动删除。

### ➤ 可用的Scrapy对象

Scrapy 终端根据下载的页面会自动创建一些方便使用的对象，例如，response 对象及 Selector 对象（对 HTML 及 XML 内容）。

这些对象有：

- `crawler`——当前 Crawler 对象。
- `spider`——处理 URL 的 Spider。
- `request`——最近获取到的页面的 request 对象。可以使用 `replace()` 修改该 request，或者使用 `fetch` 快捷方式来获取新的 request。
- `response`——包含最近获取的页面的 response 对象。
- `sel`——根据最近获取的 response 构建的 Selector 对象。
- `settings`——当前的 Scrapy settings。

这个 shell 是可以直接运行 Python 代码的，就像直接在命令行中运行 Python 一样。可以用 `print` 命令将上述对象中的成员变量打印到屏幕上。

以下是 shell 的运行效果图。





```

chinanews_crawler -- scrapy shell -- 80x24
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
'scrapy.spidermiddlewares.referer.RefererMiddleware',
'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
'scrapy.spidermiddlewares.depth.DepthMiddleware']
2017-12-28 14:48:06 [scrapy.middleware] INFO: Enabled item pipelines:
['chinanews_crawler.pipelines.BlockGamePipeline',
'chinanews_crawler.pipelines.CleanHTMLPipeline',
'chinanews_crawler.pipelines.JsonFeedPipeline']
2017-12-28 14:48:06 [scrapy.extensions.telnet] DEBUG: Telnet console listening o
n 127.0.0.1:6023
[s] Available Scrapy objects:
[s] scrapy scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s] crawler <scrapy.crawler.Crawler object at 0x10c022990>
[s] item {}
[s] settings <scrapy.settings.Settings object at 0x10c022b10>
[s] Useful shortcuts:
[s] fetch(url[, redirect=True]) Fetch URL and update local objects (by default
, redirects are followed)
[s] fetch(req) Fetch a scrapy.Request and update local object
s
[s] shelp() Shell help (print this help)
[s] view(response) View response in a browser
>>>

```

打印 request 的 URL，如下图所示。

```

chinanews_crawler -- scrapy shell -- 80x24
['chinanews_crawler.pipelines.BlockGamePipeline',
'chinanews_crawler.pipelines.CleanHTMLPipeline',
'chinanews_crawler.pipelines.JsonFeedPipeline']
2017-12-28 14:50:38 [scrapy.extensions.telnet] DEBUG: Telnet console listening o
n 127.0.0.1:6023
[s] Available Scrapy objects:
[s] scrapy scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s] crawler <scrapy.crawler.Crawler object at 0x10b977990>
[s] item {}
[s] settings <scrapy.settings.Settings object at 0x10b977b10>
[s] Useful shortcuts:
[s] fetch(url[, redirect=True]) Fetch URL and update local objects (by default
, redirects are followed)
[s] fetch(req) Fetch a scrapy.Request and update local object
s
[s] shelp() Shell help (print this help)
[s] view(response) View response in a browser
>>> fetch("http://www.baidu.com")
2017-12-28 14:50:51 [scrapy.core.engine] INFO: Spider opened
2017-12-28 14:50:51 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://www.ba
idu.com> (referer: None)
>>> print request.url
http://www.baidu.com
>>> print response.body

```

shell 还提供了一个极为强大的功能，也是我最喜欢它的一个功能（没有之一），那就是运行断点。这个功能需要在代码中植入 `inspect_response` 方法，然后运行 `scrapy crawl <蜘蛛名>` 命令，当系统执行 `inspect_response` 方法后，整个爬虫进程就会被中断，并且自动打开 Scrapy shell，此时就可以在 shell 中直接输出当前运行代码中的一些变量来进行检查。具体做法如下：

```
from scrapy.shell import inspect_response
```



```
def parse(self, response):
    # ...
    inspect_response(response, self) # 加入断点
```

```
chinanews_crawler -- scrapy crawl chinanews -- 80x24
'scrappy.spidermiddlewares.urllength.UrlLengthMiddleware',
'scrappy.spidermiddlewares.depth.DepthMiddleware']
2017-12-28 14:55:55 [scrappy.middleware] INFO: Enabled item pipelines:
['chinanews_crawler.pipelines.BlockGamePipeline',
'chinanews_crawler.pipelines.CleanHTMLPipeline',
'chinanews_crawler.pipelines.JsonFeedPipeline']
2017-12-28 14:55:55 [scrappy.core.engine] INFO: Spider opened
2017-12-28 14:55:55 [scrappy.extensions.logstats] INFO: Crawled 0 pages (at 0 pag
es/min), scraped 0 items (at 0 items/min)
2017-12-28 14:55:55 [scrappy.extensions.telnet] DEBUG: Telnet console listening o
n 127.0.0.1:6023
2017-12-28 14:55:57 [scrappy.core.engine] DEBUG: Crawled (200) <GET http://www.ch
inanews.com/rss/rss_2.html> (referer: None)
[s] Available Scrapy objects:
[s] scrapy scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s] crawler <scrappy.crawler.Crawler object at 0x110037890>
[s] item {}
[s] request <GET http://www.chinanews.com/rss/rss_2.html>
[s] response <200 http://www.chinanews.com/rss/rss_2.html>
[s] settings <scrappy.settings.Settings object at 0x110037910>
[s] Useful shortcuts:
[s] shelp() Shell help (print this help)
[s] view(response) View response in a browser
>>>
```

这个方法非常有用，尤其是当系统部署到服务器后，可以通过终端远程连入服务器进行运行期调试操作。

## 4.2.4 调试内存溢出

在 Scrapy 中，类似 requests、response 及 Items 的对象具有有限的生命周期：它们被创建、使用，最后被销毁。在这些对象中，request 的生命周期应该是最长的，其会在调度队列（Scheduler queue）中一直等待，直到被处理。

由于这些 Scrapy 对象拥有很长的生命周期，因此将这些对象存储在内存中而没有正确释放的危险总是存在。这也导致了所谓的“内存泄漏”。为了帮助调试内存泄漏，Scrapy 提供了跟踪对象引用的机制，叫作 trackref，也可以使用第三方提供的更先进内存调试库 Guppy。而这些都是都必须在 Telnet 终端中使用。

### 内存泄漏的常见原因

内存泄漏经常是因为 Scrapy 开发者在 requests 中（有意或无意）传递对象的引用（例如，使用 meta 属性或 request 回调函数），使得该对象的生命周期与 request 的生命周期绑定。这是到目前为止最常见的内存泄漏的原因，同时对新手来说也是一个比较难调试的问题。





在大型项目中, Spider 是由不同的人编写的。其中有的 Spider 可能是有“泄漏的”, 当所有爬虫同时运行时, 这些 Spider 影响了其他(写好)的爬虫, 最终影响了整个爬取进程。

与此同时, 在不限制框架功能的同时, 避免造成泄漏是十分困难的。因此, 我们决定不限制这些功能, 而是提供调试这些泄漏的实用工具。这些工具回答了一个问题: 哪个 Spider 在发生泄漏。

内存泄漏可能存在与一个编写的中间件、管道(pipeline)或扩展, 以及在代码中没有正确释放(之前分配的)资源有关。例如, 在 spider\_opened 中分配资源但没有在 spider\_closed 中释放它们。

### 使用trackref调试内存泄漏

trackref 是 Scrapy 提供的用于调试大部分内存泄漏情况的模块。简单来说, 它追踪了所有活跃(live)的 response、request、Item 及 Selector 对象的引用。

可以进入 telnet 终端并通过 prefs() 功能来检查有多少(上面提到的)活跃(live)对象。pref() 是 print\_live\_refs() 功能的引用:

```
telnet localhost 6023
```

```
>>> prefs()
```

```
Live References
```

ExampleSpider	1	oldest: 15s ago
HtmlResponse	10	oldest: 1s ago
Selector	2	oldest: 0s ago
FormRequest	878	oldest: 7s ago

报告也展现了每个类中最老的对象的时间(age)。

如果有内存泄漏, 则查看最老的 request 或 response 就是能找到哪个 Spider 正在泄露的机会。可以使用 get\_oldest()方法来获取每个类中最老的对象(在终端中)。

### 哪些对象被追踪了

trackref 追踪的对象包括以下类(及其子类)的对象:

- scrapy.http.Request
- scrapy.http.Response
- scrapy.item.Item



- scrapy.selector.Selector
- scrapy.spider.Spider

### 案例

让我们来看一个假设具有内存泄漏的例子。

假如有些 Spider 的代码中有一行类似于这样的代码：

```
return Request("http://www.somenastyspider.com/product.php?pid=%d" % product_id,
               callback=self.parse, meta={referer: response})
```

在 request 中传递了一个 response 的引用,使得 response 的生命周期与 request 绑定,进而造成了内存泄漏。

让我们来看看如何使用 trackref 工具来发现哪一个是有问题的 Spider(当然是在不知道任何前提的情况下)。

当 crawler 运行了一段时间后,我们发现内存占用增长了很多。这时进入 Telnet 终端,查看活跃(live)的引用:

```
>>> prefs()
Live References

SomenastySpider          1  oldest: 15s ago
HtmlResponse             3890 oldest: 265s ago
Selector                  2  oldest: 0s ago
Request                   3878 oldest: 250s ago
```

上面具有非常多活跃(且运行时间很长)的 response,而其比 request 的时间还要长的现象肯定是有问题的。因此,查看最老的 response:

```
>>> from scrapy.utils.trackref import get_oldest
>>> r = get_oldest('HtmlResponse')
>>> r.url
'http://www.somenastyspider.com/product.php?pid=123'
```

通过查看最老的 response 的 URL,我们发现其属于 somenastyspider.com spider。现在我们可以查看该 Spider 的代码并发现导致泄漏的那行代码(在 request 中传递 response 的引用)。

如果想要遍历所有而不是最老的对象,则可以使用 iter\_all()方法:





```
>>> from scrapy.utils.trackref import iter_all
>>> [r.url for r in iter_all('HtmlResponse')]
['http://www.somenastyspider.com/product.php?pid=123',
 'http://www.somenastyspider.com/product.php?pid=584',
 ...]
```

## 很多Spider

如果项目中有很多的 Spider，则 `prefs()` 的输出会变得很难阅读。针对此问题，该方法具有 `ignore` 参数，用于忽略特定的类（及其子类）。例如：

```
>>> from scrapy.spider import Spider
>>> prefs(ignore=Spider)
```

这时不会展现任何 Spider 的活跃引用。

### ➤ scrapy.utils.trackref 模块

以下是 `trackref` 模块中可用的方法。

```
class scrapy.utils.trackref.object_ref
```

如果想通过 `trackref` 模块追踪活跃的实例，则继承该类（而不是对象）：

```
scrapy.utils.trackref.print_live_refs(class_name, ignore=NoneType)
```

打印活跃引用的报告，以类名分类。

参数：

- `ignore`（类或者类的元组）——如果给定，则所有指定类（或者类的元组）的对象将被忽略。
- `scrapy.utils.trackref.get_oldest(class_name)`——返回给定类名的最老活跃（live）对象，如果没有则返回 `None`。

首先使用 `print_live_refs()` 来获取每个类所跟踪的所有活跃（live）对象的列表。

```
scrapy.utils.trackref.iter_all(class_name)
```

返回一个能给定类名的所有活跃对象的迭代器，如果没有则返回 `None`。首先使用 `print_live_refs()` 来获取每个类所跟踪的所有活跃（live）对象的列表。

## 使用Guppy调试内存泄漏

`trackref` 提供了追踪内存泄漏非常方便的机制，其仅仅追踪了可能导致内存泄漏的对象



(requests、response、Items 及 Selectors)。然而，内存泄漏也有可能来自其他（更为隐蔽的）对象。如果是因为这个原因，通过 `trackref` 则无法找到泄漏点，不过仍然有其他工具可以实现：Guppy library。

如果使用 `setuptools`，则可以通过下列命令安装 Guppy：

```
$ easy_install guppy
```

Telnet 终端也提供了快捷方式 (`hpy`) 来访问 Guppy 堆对象 (heap objects)。下面给出了查看堆中所有可用的 Python 对象的例子：

```
>>> x = hpy.heap()
>>> x.bytype
Partition of a set of 297033 objects. Total size = 52587824 bytes.
Index  Count   %      Size  % Cumulative  % Type
   0    22307   8  16423880  31  16423880  31 dict
   1   122285  41  12441544  24  28865424  55 str
   2    68346  23   5966696  11  34832120  66 tuple
   3      227   0   5836528  11  40668648  77 unicode
   4     2461   1   2222272   4  42890920  82 type
   5    16870   6   2024400   4  44915320  85 function
   6    13949   5   1673880   3  46589200  89 types.CodeType
   7    13422   5   1653104   3  48242304  92 list
   8     3735   1   1173680   2  49415984  94 _sre.SRE_Pattern
   9     1209   0    456936   1  49872920  95 scrapy.http.headers.Headers
<1676 more rows. Type e.g. '_more' to view.>
```

可以看到大部分的空间被字典所使用。查看哪些属性引用了这些字典：

```
>>> x.bytype[0].byvia
Partition of a set of 22307 objects. Total size = 16423880 bytes.
Index  Count   %      Size  % Cumulative  % Referred Via:
   0   10982  49   9416336  57   9416336  57 '.__dict__'
   1    1820   8   2681504  16  12097840  74 '.__dict__', '.func_globals'
   2     3097  14   1122904   7  13220744  80
   3     990   4    277200   2  13497944  82 "['cookies']"
   4     987   4    276360   2  13774304  84 "['cache']"
   5     985   4    275800   2  14050104  86 "['meta']"
   6     897   4    251160   2  14301264  87 '[2]'
```





```
7      1    0  196888    1  14498152  88 "['moduleDict']", "['modules']"  
8    672    3  188160    1  14686312  89 "['cb_kwargs']"  
9     27    0  155016    1  14841328  90 '[1]'  
<333 more rows. Type e.g. '_.more' to view.>
```

如上所示，Guppy 模块十分强大，不过也需要一些关于 Python 内部的知识。

有时候，我们可能会注意到 Scrapy 进程的内存占用只在增长，从不下降。这并不是 Scrapy 或者项目在泄漏内存，这是由于一个已知（但不有名）的 Python 问题。

这个 patch 仅仅会释放没有在其内部分配对象的区域（arena）。这意味着碎片化是一个大问题。某个应用可以拥有很多空闲内存，分布在所有的区域（arena）中，但是无法释放任何一个。这个问题存在于所有内存分配器中。解决这个问题的唯一办法是转化到一个更为紧凑（compact）的垃圾回收器中，其能在内存中移动对象。这需要对 Python 解析器做一个显著的修改。

## 4.3 处理HTTP请求

在讲述第一个入门例子时用了很少的篇幅简单地说明了 HTTP 的基本原理，力求以最平滑的学习曲线进入虫术的“状态”中。而在中级虫术中，我们之所学习 HTTP，是因为有可能将其应用于真正的项目实践中。当我们将所谓深奥的技术简化为其本质的内容时，其实都是一些很简单的基本知识。它们之所以深奥，是被多次迭代与实践后才形成的，万法归一，掌握本质则可触类旁通，这也是我多年来学习各类技术的一点心得。

因此本节将是全书的一个技术转折点，极为重要。吃透了 HTTP 就揭开了网页开发的本质，即使不是做互联网前后端开发的读者，也可以从另一个方面来理解前端开发与后端开发的本质，因为它们最终都要遵循 HTTP 的规则，遵守 HTML 的规范。无论是 Java、C#、Ruby，还是 Python、PHP，只要从 HTTP 和 HTML 的角度看都是一样的，最终还是获得一个标准的 HTTP 请求，返回一个 HTML 规范的响应内容，只是实现手段有所差异而已。

HTTP 是一种基于 TCP/IP 的通信协议，它的默认通信端口为 80。它的通信原理是客户端与指定 URL 的主机连接后向主机发出一个协议报文，在此报文内会带有客户端的相关信息。例如，身份信息、客户端信息、客户端地址或者带有数据的表单等，提供这些信息是为了告知主机需要从服务器上获取什么资源，这个报文我们称之为 HTTP 请求；当主机接收到来自客户端的 HTTP 请求后，就会按照请求报文的内容生成相应的结果，可能是文字或者二进制流数据，然后生成一个新的结果性报文并返回给客户端，这个结果性的报文则是服务器响应。正是基于这个简单的超文本传输协议，才有了如今多姿多彩的互联网生态。

只要记住一点，请求与响应是“孪生兄弟”，它们任何一个单独存在都没有意义，一定是先



有请求再有响应。下面会将 HTTP 协议结合 Scrapy 提供的请求对象 request 和 response 来深入解释它们的意义与作用。这对孪生兄弟是爬虫存在的根本，因此很有必要了解它们的全貌并知道如何在爬虫中使用。

### 4.3.1 HTTP请求

超文本传输协议（Hypertext Transfer Protocol，简称 HTTP）是应用层协议。HTTP 是一种请求/响应式的协议，即一个客户端与服务器建立连接后，向服务器发送一个请求；服务器接到请求后，给予相应的响应信息。

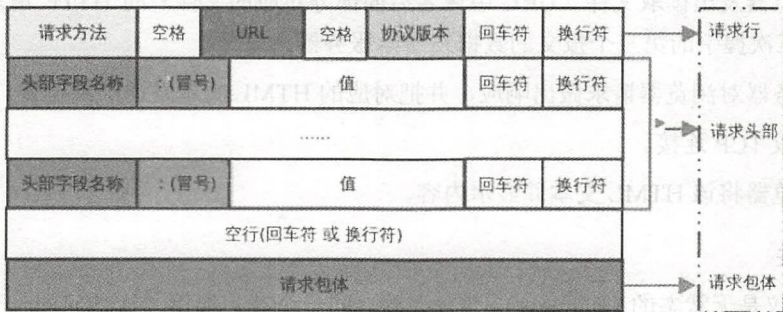
HTTP 请求是一种人机可读明文，以百度为例，在浏览器上直接打开百度后可以从浏览器的开发者模式中看到以下网络请求内容：

```
GET http://www.baidu.com HTTP/1.1
```

通常 HTTP 消息包括客户机向服务器的请求消息和服务器向客户机的响应消息。这两种类型的消息由一个起始行、一个或者多个头域、一个只是头域结束的空行和可选的消息体组成。HTTP 的头域包括通用头、请求头、响应头和实体头四个部分。每个头域由一个域名、冒号（:）和域值三部分组成。域名是大小写无关的，域值前可以添加任何数量的空格符，头域可以被扩展为多行，在每行开始处，使用至少一个空格或制表符。

我并不打算将 HTTP 协议的所有内容直接罗列出来，协议说明和类手册一样无趣难懂，甚至会让你一下子失去对它的兴趣。虽然在将来你还是会仔细地阅读它，但我认为学习任何新的技术，在入手阶段最重要的是培养自己对这门技术的兴趣。因此，先不管具体的协议内容，只要知道 HTTP 请求就是向某一个服务器地址或者更确切地说是某一个具体的网络资源（URI）指定一种执行方法（HTTP 方法）并返回结果给我们就可以了。

HTTP 协议原理如下图所示。



- 请求头部：请求头部由关键字/值对组成，每行一对，关键字和值用英文冒号“:”分隔。





请求头部通知服务器有关客户端请求的信息。

- 空行: 最后一个请求头之后是一个空行, 发送回车符和换行符, 通知服务器以下不再有请求头。
- 请求包体: 请求包体不在 GET 方法中使用, 而是在 POST 方法中使用。POST 方法适用需要客户填写表单的场合。与请求包体相关的常用的是包体类型 Content-Type 和包体长度 Content-Length。

## HTTP的工作原理

HTTP 协议采用请求/响应模型。客户端向服务器发送一个请求报文, 服务器以一个状态作为响应。

以下是 HTTP 请求/响应的步骤。

- 客户端连接到 Web 服务器: HTTP 客户端与 Web 服务器建立一个 TCP 连接。
- 客户端向服务器发起 HTTP 请求: 通过已建立的 TCP 连接, 客户端向服务器发送一个请求报文。
- 服务器接收 HTTP 请求并返回 HTTP 响应: 服务器解析请求, 定位请求资源, 服务器将资源副本写到 TCP 连接, 由客户端读取。
- 释放 TCP 连接: 若 connection 模式为 close, 则服务器主动关闭 TCP 连接, 客户端被动关闭连接, 释放 TCP 连接; 若 connection 模式为 keepalive, 则该连接会保持一段时间, 在该时间内可以继续接收请求。
- 客户端浏览器解析 HTML 内容: 客户端将服务器响应的 HTML 文本解析并显示。

例如, 在浏览器地址栏键入 URL, 按下回车键后会经历以下流程:

- (1) 浏览器向 DNS 服务器请求解析该 URL 中域名所对应的 IP 地址。
- (2) 解析出 IP 地址后, 根据该 IP 地址和默认端口 80, 同服务器建立 TCP 连接。
- (3) 浏览器发出读取文件 (URL 中域名后面部分对应的文件) 的 HTTP 请求, 该请求报文作为 TCP 三次握手的第三个报文的数据发送给服务器。
- (4) 服务器对浏览器请求做出响应, 并把对应的 HTML 文本发送给浏览器。
- (5) 释放 TCP 连接。
- (6) 浏览器将该 HTML 文本并显示内容。

## 无状态性

HTTP 协议是无状态的 (stateless)。也就是说, 同一个客户端第二次访问同一个服务器上的页面时, 服务器无法知道这个客户端曾经访问过, 服务器也无法分辨不同的客户端。HTTP 的



无状态特性简化了服务器的设计，使服务器更容易支持大量并发的 HTTP 请求。

### HTTP持久连接

HTTP 1.0 使用的是非持久连接，主要缺点是客户端必须为每一个待请求的对象建立并维护一个新的连接，即每请求一个文档就要有两倍 RTT 的开销。因为同一个页面可能存在多个对象，所以非持久连接可能使一个页面的下载变得十分缓慢，而且这种短连接增加了网络传输的负担。HTTP 1.1 使用持久连接 **keep-alive**，所谓持久连接，就是服务器在发送响应后仍然在一段时间内保持这条连接，允许在同一个连接中存在多次数据请求和响应，即在持久连接情况下，服务器在发送完响应后并不关闭 TCP 连接，而客户端可以通过这个连接继续请求其他对象。

HTTP/1.1 协议的持久连接有两种方式。

- 非流水线方式：客户在收到前一个响应后才能发出下一个请求。
- 流水线方式：客户在收到 HTTP 的响应报文之前就能接着发送新的请求报文。

具体例子：

```
Remote Address:116.57.254.104:80
Request URL:http://hr.tencent.com/
Request Method:GET
Status Code:200 OK

Request Headers
GET / HTTP/1.1
Host: hr.tencent.com
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/35.0.1916.114 Safari/537.36
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,zh-CN;q=0.6,zh;q=0.4
Cookie: pgv_pvi=2098703360; PHPSESSID=bc7onl0dojbsatcsfv79pds77; pgv_info=ssid=s1454606128; pgv_pvid=926725350; ts_uid=4084753309

Response Header
HTTP/1.1 200 OK
Server: nginx
```





```
Date: Mon, 26 Jan 2015 01:09:10 GMT
Content-Type: text/html;charset=utf-8
Content-Length: 3631
Connection: keep-alive
X-Powered-By: PHP/5.3.10
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Vary: Accept-Encoding
Content-Encoding: gzip
```

从请求报文可以知道：

```
GET / HTTP/1.1
```

请求方法 GET 表示一个读取请求，从服务器获得网页数据，“/”表示 URL 的路径，URL 总是以 “/” 开头，“/” 就表示首页，最后的 HTTP/1.1 指采用的 HTTP 协议版本是 1.1。请求域名如下所示。

```
Host: hr.tencent.com
```

响应报文如下：

```
HTTP/1.1 200 OK
Server: nginx
```

## HTTP方法

HTTP 方法是告知服务器应该对客户端请求的这个 URL 地址上的资源执行一个什么样的动作。

常见的方法：

- GET——浏览器告知服务器只获取页面上的信息并发给浏览器。这是一种只读行为，也是最常用的方法。
- POST——浏览器告诉服务器想在指定的 URL 上发布新信息。并且服务器必须确保数据已存储且仅存储一次。这是 HTML 表单通常发送数据到服务器的方法。这是一种“添加”/“新增”行为。
- PUT——类似 POST，但是服务器可能触发了多次存储过程，多次覆盖旧值。你可能会



问这有什么用，当然这是有原因的。考虑到传输中连接可能会丢失，在这种情况下，浏览器和服务端之间的系统可能安全地进行第二次接收请求，而不破坏其他东西。因为 POST 只触发一次，所以用 POST 是不可能的。这是一种“更新”行为。

- DELETE——删除给定位置的信息。这是“删除行为”。
- HEAD——浏览器告诉服务器欲获取信息，但是只关心消息头。服务器端会像处理 GET 请求一样来处理它，但是不分发实际内容。
- OPTIONS——给客户端提供一个敏捷的途径来弄清这个 URL 支持哪些 HTTP 方法。

GET、POST、PUT 和 DELETE 属于常用的四大 HTTP 方法，而 HEAD 和 OPTIONS 则属于对 GET 的一种延伸，在 Web 编程中极少使用，它们更多地被浏览器使用，这正是它们会在此被提及的原因。在后面的内容中有时需要将虫子模拟成人在使用浏览器，那么就有可能使用到这两种“隐性”的 HTTP 方法。

无论用什么语言实现 HTTP 请求，都离不开以上理论。在初阶虫术中就采用了 urllib 来演示虫的本质。urllib 作为 Python 的内置 HTTP 库，虽然是一个不错的便捷选择，但其易用性还有待改进，因此本节中改用了另一个更为出名的 Python 库：**requests**([http://cn.python-requests.org/zh\\_CN/latest/index.html](http://cn.python-requests.org/zh_CN/latest/index.html))。

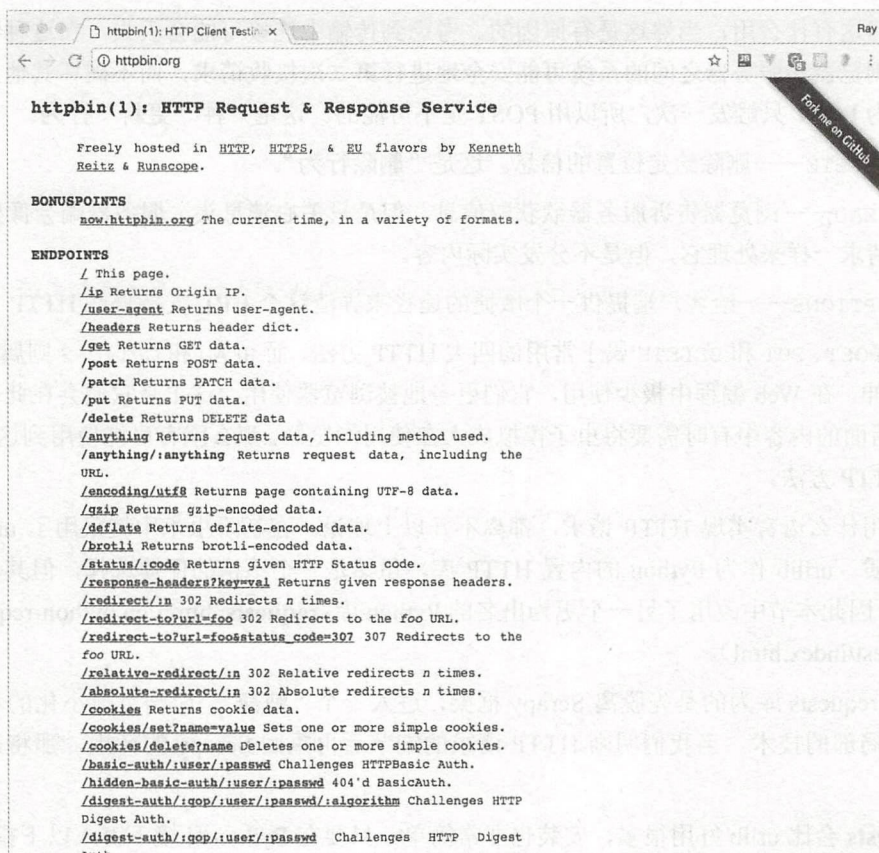
使用 requests 库为的是先脱离 Scrapy 框架，进入一个“纯净”、简洁且最小化的环境中学习与实践局部的技术。当我们明晰 HTTP 请求的细节后再重回 Scrapy 的怀抱，那将是另一番的景象。

**requests** 会比 urllib 好用很多，安装也非常简单，只要在 Python 环境下键入以下指令即可：

```
$ pip install requests
```

为了达到良好的示例效果，接下来采用 <http://httpbin.org/> 网站提供的 URI 来进行演示。httpbin 提供了一系列 URL 给用户进行 HTTP 请求的相关测试，它最好的地方是将服务端所接收到的 request 信息以 JSON 的形式返回给我们作为参考，如下图所示。





先用 requests 库向 httpbin 上的一个示例网页地址发送一个 Get 请求（相当于用浏览器直接打开这个地址），然后打印输出这个网页上的内容：

```
>>> import requests
>>> r = requests.get('http://httpbin.org/html')
>>> print r.text
```

运行打印指令后会看到以下输出结果：

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
```

```

<h1>Herman Melville - Moby-Dick</h1>
<div>
  <p>
    Availing himself of the mild, ....
    这里是一堆很长的英文，具体含义对本书毫无意义，只作为一种占位符用，因此省去
  </p>
</div>
</body>
</html>

```

由上述代码可知，requests 以 HTTP 方法来包装它的方法函数，这样更便于我们记忆与使用，其他 HTTP 请求类型 PUT、DELETE、HEAD 及 OPTIONS 又是如何的呢？也都是一样的简单：

```

>>> r = requests.post("http://httpbin.org/post")
>>> r = requests.put("http://httpbin.org/put")
>>> r = requests.delete("http://httpbin.org/delete")
>>> r = requests.head("http://httpbin.org/get")
>>> r = requests.options("http://httpbin.org/get")

```

## 参数化页面

不同的参数会直接影响网页的输出，这是服务端页面的特质。不过随着前端技术的发展，这一参数化的特性近年来也被引入到了不少知名的前端框架内，例如，ReactJS、AngularJS 和 Vue 等（我的另一本书《Vue2 实践揭秘》中也有相关内容的介绍）。参数化页面多用于 GET 方法，但实际上它没有被 HTTP 方法所限制，使用任何一种 HTTP 方法服务端都可以得到这些参数，因为这些参数就是 URL 的一部分。

这种参数化主要分为两种表现形式。

### ➤ 第一：查询字符串

就是在 URL 后加上“？”，然后以 Key=Value&Key1=Value1&...&KeyN=valueN 的方式传递参数。这种方式传递的参数（长度）都会比较小，是最传统的一种参数传递方式。但由于这种形式的 URL 并不便于记忆，也违反了人机可读这一要求，在 Web 技术运用得比较好的一些网站和开发团队中已经慢慢被淘汰，或者说只作为某些特殊场合的一种补充。

举一个常见的博客文章的 URL 地址例子：

```
http://www.examples.com/blogs?post=1
```



这个地址看起来是不是有点熟悉? 将 `post=N` 后的值换成不同的整数就可以显示不同的博客内容, 这是一些技术陈旧的网站经常采用的一种方式, 这一点在编写爬虫前进行网页分析时是必须要留意的。

如果用 `requests` 发起一个带有查询字符串的地址服务端, 会收到什么样的内容呢?

```
>>> r = requests.get('http://httpbin.org/get?post=100')
>>> print r.text
{
  "args": {
    "post": "100"
  },
  // ... 此处内容省略
}
```

如果我们用 Python、ROR (Ruby On Rails) 或者 Node.js (Express) 这些语言作为后端服务器之用, 则可以从它们所提供的服务端请求对象中读取 `args` 参数来获取查询参数。

`.net framework` (ASP.net) 可以从 `QueryString` 字典属性中获取。

这些都是一种我们必须要知道的通用规则, 因为虫子总得知道要爬向哪个网页的内容, 带参网页往往是没有数量边界的, “聪明”的虫子甚至会对这些没边的页面在爬取之前行预测。

## ➤ 第二: 路由

路由是近十年来对参数化页面进行改进后的一种 URL 方式, 路由技术都是通过 Web 开发框架提供的。通过外部的单一 URL 是很难判断这是一个路由参数化的页面。以上面的代码为例, 用一个良好的路由规则来生成页面:

```
http://www.examples.com/blogs/how-to-write-a-spider.html
```

这种地址更像是一个静态文件。因为这样的地址更受搜索引擎的青睐, 会在搜索引擎的 PV 判定中得到相当高的分数, 所以不少开发人员甚至开发框架都会采用这种隐式化的 URL 构建路由规则。

由于它们的处理在 HTTP 请求这一内容中都是相同的, 所以没有必要展开。在高级虫术中会讲述如何写一些聪明的虫子去判断哪些地址是路由生成的, 甚至可以逆向推导这些路由的生成规则。

## 提交表单

表单通过 POST 或者 PUT 方法将网页上<form>元素中的输入域全部提交到<form>元素所指向的地址上,通过提交表单可以突破数据量小的局限,因为表单可以提交任何内容,包括各种二进制文件。但 POST 和 PUT 两种 HTTP 方法会给服务端带来较大的负荷,而且 POST 后的内容可能与开始访问的网页内容是完全不同的,这样爬虫就得进行额外的处理。这些内容会在下文的处理表单一节中用具体的应用示例说明,此处还是先将关注点移到 HTTP 请求上。

可以用 requests 产生一个同时带有查询字符串的 URL 和用 GET 方法提交表单的 URL,看看服务端会接收到什么:

```
>>> r = requests.post('http://httpbin.org/post?q=query+data', data =
{'terms': 'Here is test'})
>>> print r.text
{
  "args": {
    "q": "query data"
  },
  "data": "",
  "files": {},
  "form": {
    "terms": "Here is test"
  },
  "headers": {
    "Accept": "*//*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Content-Length": "18",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.2"
  },
  "json": null,
  "origin": "117.136.40.160",
  "url": "http://httpbin.org/post?q=query+data"
}
```

如果在服务端采用 Flask 或者 Django 这类 Python 服务端框架来接收以上请求,则会得到一个几乎与上述代码相同的服务端 request 对象。通过这样一种穿透性的“透视”实验,我们可以



很清楚地知道发出的请求最终在服务端会呈现什么。

HTTP request 在发送到远程主机之前会生成一种 HTTP 协议规定的文本, 这里不会讲述原生的 HTTP 请求到底长什么样, 因为爬虫根本不需要去了解它, 我们只要在对象上知道请求怎么使用就够了。

查询字符串会被放到服务端请求对象的 args 数组里面(.NET 会在放置在 QueryString 对象中), 而表单域中的数据则被放到 form 属性对象中。如果表单中带有

请求头

如果说查询字符串和表单是客户端与服务端之间的显式参数传递, 那么请求头则是一种隐式的参数传递方式。请求头实际上就是一组附在请求对象上的“键-值”(Key-Value) 对。请求头内附带的信息一般来说都是向服务端描述客户端“能力”的内容。

通过设置不同的请求头域可以进行客户端伪装、配置代理、设置明文登录等功能。

下表为常用头域。

头 域	解 释	示 例
Accept	客户端可识别的响应内容类型列表; 星号 “*” 用于按范围将类型分组, 用 “/” 指示可接收全部类型, 用 “type/*” 指示可接收 type 类型的所有子类型	Accept: text/plain, text/html
Accept-Charset	浏览器可以接收的字符编码集	Accept-Charset: iso-8859-5
Accept-Encoding	指定浏览器可以支持的 Web 服务器返回内容压缩编码类型	Accept-Encoding: compress, gzip
Accept-Language	浏览器可接收的自然语言	Accept-Language: en,zh
Accept-Ranges	可以请求网页实体的一个或者多个子范围字段	Accept-Ranges: bytes
Authorization	HTTP 授权的授权证书	Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Cache-Control	指定请求和响应遵循的缓存机制	Cache-Control: no-cache
Connection	表示是否需要持久连接(close 或 keepalive)	(HTTP 1.1 默认进行持久连接) Connection: close

续表

头 域	解 释	示 例
Cookie	HTTP 请求发送时, 会把保存在该请求域名下的所有 Cookie 值一起发送给 Web 服务器	Cookie: \$Version=1; Skin=new;
Content-Length	请求的内容长度	Content-Length: 348
Content-Type	请求的与实体对应的 MIME 信息	Content-Type: application/x-www-form-urlencoded
Date	请求发送的日期和时间	Date: Tue, 15 Nov 2010 08:12:31 GMT
Expect	请求的特定的服务器行为	Expect: 100-continue
From	发出请求的用户的 E-mail	From: user@email.com
Host	指定请求的服务器的域名和端口号	Host: www.zcmhi.com
If-Match	只有请求内容与实体相匹配才有效	If-Match: "737060cd8c284d8af7ad3082f209582d"
If-Modified-Since	如果请求的部分在指定时间之后被修改则请求成功, 未被修改则返回 304 代码	If-Modified-Since: Sat, 29 Oct 2010 19:43:31 GMT
If-None-Match	如果内容未改变返回 304 代码, 参数为服务器先前发送的 Etag, 与服务器回应的 Etag 比较判断是否改变	If-None-Match: "737060cd8c284d8af7ad3082f209582d"
If-Range	如果实体未改变, 则服务器发送客户端丢失的部分, 否则发送整个实体。参数也为 Etag If-Range: "737060cd8c284d8af7ad3082f209582d"	
If-Unmodified-Since	只在实体在指定时间之后未被修改才请求成功	If-Unmodified-Since: Sat, 29 Oct 2010 19:43:31 GMT
Max-Forwards	限制信息通过代理和网关传送的时间	Max-Forwards: 10
Pragma	用来包含实现特定的指令	Pragma: no-cache
Proxy-Authorization	连接到代理的授权证书	Proxy-Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Range	只请求实体的一部分, 指定范围	Range: bytes=500-999
Referer	先前网页的地址, 当前请求网页紧随其后, 即来路	Referer: http://www.dotnetage.com/archives/1.html



续表

头 域	解 释	示 例
TE	客户端愿意接收的传输编码，并通知服务器接收尾加头信息	TE: trailers,deflate;q=0.5
Upgrade	向服务器指定某种传输协议以便服务器进行转换（如果支持）	Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11
User-Agent	产生请求的浏览器类型	User-Agent: Mozilla/5.0 (Linux; X11)
Via	通知中间网关或代理服务器地址，通信协议	Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
Warning	关于消息实体的警告信息	Warn: 199 Miscellaneous warning

4.3.2 Scrapy的Request对象

接下来我们就需要知道在 Scrapy 中如何处理请求,在前面例子中已经接触过 request 对象,这是由 Scrapy 提供的一个 HTTP 请求对象。它的使用也非常简单，我们并不能像 urllib 或者 requests 那样直接操作它，因为它在 Scrapy 中仅仅是一个数据载体，真正向远程主机发出请求的是下载器。

以下是 Request 类的构造函数及其参考说明：

```
class scrapy.http.Request(url[, callback, method='GET', headers, body, cookies, meta, encoding='utf-8', priority=0, dont_filter=False, errback])
```

参 数	类 型	说 明
url	string	目标请求地址
callback	callable	执行请求后当远程主机产生响应并由下载器生成响应对象后调用的自定义回调函数，如果不指定下载器，则自动调用蜘蛛上的 parse 方法并传入 response 对象
method	string	指定 HTTP 方法，默认为 GET
meta	dict	Request.meta 属性内初始化值，下文中有详解
body	str 或 unicode	请求的正文，一般情况下我们没有必要使用它，下文会用其他办法来处理需要使用请求正文的情况
headers	dict	HTTP 请求头
cookies	dict 或 list	附带在请求中一起发出的 Cookies 对象
encoding	string	当前请求的编码方式（默认为 UTF-8）
priority	int	设置请求的优先级（默认为 0）。这个优先级是 scheduler 在进程中用于定义处理请求的顺序

续表

参 数	类 型	说 明
dont_filter	boolean	标记当前请求在 scheduler 内不被过滤。当多次处理一个独立自定义请求时，可以设置此参数以避免 scheduler 将请求进行冗余过滤。在使用时一定要非常小心，因为有可能会进入一个请求的死循环中
errback	callable	当处理请求发生任何异常时就会调用此回调函数。包括页面处理遇到最常见的 404 错误等。它会返回一个 Twisted 的错误实例作为函数的第一个参数

### 示例 1：全面遍历

在了解了“通用蜘蛛”之后，应该对蜘蛛的实现方式与原理有相当清楚的认识，并且通过“深入 Spider”一节，以及对 Spider 的代码分析，相信现在的你对 Spider 已经有了更深层次的理解，那么现在尝试跳过 CrawlSpider 来写一个具有深度遍历能力的蜘蛛以了解 request 的用法，代码如下：

```
from scrapy import Spider
from scrapy.linkextractor import LinkExtractor

class DeepInSpider(Spider):
    name = 'example.com'
    start_urls = ['http://www.example.com/default']

    def parse(self, response):
        link_extractor = LinkExtractor()
        seen = set() # set 是 Python 中的一种不重复的数据集合
        # 此处是为了记录哪些页面已经被爬过了，避免重复爬取相同的页面内容
        links = [l for l in link_extractor.extract_links(response) if l not in seen]
        for link in links:
            seen.add(link)
            cb = None
            if (link.contains('detail')):
                cb = parse_detail
            yield Request(url=link, callback=cb)

    def parse_detail(self, response):
        # 分析详细网页内容
```



Pass

以上代码的逻辑很简单, 首先由 Spider 基类通过 `start_urls` 指定的地址生成“首爬”请求, 这里模拟的就是一个默认页面。这种情况是很普遍的, 可适用于那些我们根本不知道页面深度与起始页面位置, 但只知道其中一个页面就是我们要找的数据的场景(这里就是在 URL 中带有 `detail` 字符串的页面)。可以像本例一样先从首页开始进行广度遍历, 如果没有发现目标网址, 则产生第二批网页请求。不知道读者有没有发现这个例子是含有隐性循环的, 深度遍历就是利用这种隐形循环进行的。

当 `request` 的 `callback` 参数设置为 `None` 时, 这个请求在下载器完成处理后还是会回到当前的 `parse` 方法调用中, 在代码中虽没有循环, 但却在类的协作中实现了这种递归。而 `request` 的处理出口就在于对回调函数设定为 `parse_detail`。同理, 如果在 `parse_detail` 中返回的是 `Item` 而不再是新的 `request`, 那么当前的递归就会宣告结束。

**注意:** 不要轻易将以上示例应用于具有反爬机制的网站, 否则非常容易被封 IP。

## 示例 2: 批量生成爬网起始请求与随机客户端模拟

这个示例会包含一些打算在高级虫术中才讲述的内容, 在此也作为一个引子, 希望能起到抛砖引玉的效果。这个例子也是从实践中得来的, 是针对博客中的文章进行爬网的一个示例。

它的特殊性在于:

- 没有综合性的起始入口;
- 无法准确地得知博客中到底有多少篇文章;
- 要具有一定的隐蔽性。

```
from scrapy import Spider
from random import randint

class PostSpider(Spider):
    name = 'example.com'
    post_rule = "http://www.example.com/posts/%s"
    user_agents = []

    def gen_post_urls(self, _max=2048):
        for i in range(_max):
            yield self.post_rule % i

    def start_requests(self):
```

```

for url in gen_post_urls():
    yield Request(url, headers={'User-Agent': user_agents[randint(len
(user_agents))]])

def parse(self, response):
    # 这里处理返回的博客数据
    Pass

```

实现起来并不复杂，用 `gen_post_urls` 函数生成文章的路由访问规则，设置一个上限，一次性生成我们猜测可能存在的 URL，然后在生成 `request` 时随机地设置它的请求头中的 `User-Agent` 域，以欺骗对方的主机当前的 `request` 是由哪一种浏览器发出的，虽说有点掩耳盗铃的意味，但对于很多根本没有反爬概念与防范的网站却极为有效。

#### 附：Request的实例属性与说明（见下表）

属 性	说 明
<code>headers</code>	用于存储请求头的一个与字典型的对象
<code>body</code>	请求的正文
<code>meta</code>	一个可以存储任意内容的字典对象，用于在线程与方法之间“Hold”住一些系统或自定义的变量
<code>copy()</code>	从当前 <code>request</code> 对象中复制一个实例副本
<code>replace([url, method, headers, body, cookies, meta, encoding, dont_filter, callback, errback])</code>	用指定的参数重新替换当前请求中的成员属性值

#### ► FormRequest对象

这是一个继承自 `request` 对象并专门用于处理表单请求的对象。`FormRequest` 可以从 `response` 对象中自动分析响应网页结果中的表单并对表单内的输入域进行预填充。

以下是 `FormRequest` 的构造函数：

```
class scrapy.http.FormRequest(url[, formdata, ...])
```

`FormRequest` 的使用非常简单，只要提供表单提交时的目标 URL 和表单中必备的输入域的“键-值”对的字典或元组对象即可。

例如，产生一个向 `httpbin.org` 提交查询的请求：

```
req = FormRequest('http://www.httpbin.org', formdata = {'terms': 'python'})
```



FormRequest 提供了一个非常有用的类方法 `from_response`，这是一个工厂方法，它能从下载器返回的 `response` 对象中自动识别表单并进行预填充，返回一个 `FormRequest` 实例。

```
classmethod from_response(response[, formname=None, formnumber=0,
formdata=None, formxpath=None, clickdata=None, dont_click=False, ...])
```

这个方法会自动将表单的“提交”行为模拟成“单击”，因为大多数的表单都是以单击（表单内的 `<input type="submit" />` 按钮）作为默认的提交方式。虽然这样做很方便，但有时却可能会给调试带来一些问题。例如，当使用 `JavaScript` 对表单进行填充或者提交时，`from_response()` 构建的这一默认行为就可能不适用了。要禁止这种“可单击”默认提交行为，可以设置 `dont_click` 为 `True`。

参数说明如下表所示。

名 称	类 型	说 明
<code>response</code>	<code>Response</code>	一个承载有 HTML 表单的响应对象实例，用于从此实例中读取表单域以进行预填充
<code>formname</code>	<code>string</code>	显式指定生成表单请求实例所使用的表单名称
<code>formxpath</code>	<code>string</code>	使用 XPath 所匹配的表单作为生成请求的实例
<code>formnumber</code>	<code>integer</code>	通过索引指定采用哪一表单（默认为 0）
<code>formdata</code>	<code>dict</code>	输入域使用的数据。如果在 <code>&lt;form&gt;</code> 元素中已有对应的域，那么预填充的域值就会被当前指定的值覆盖
<code>clickdata</code>	<code>dict</code>	指定被单击的控件属性。如果为空表单，则会模拟第一个元素被单击
<code>dont_click</code>	<code>boolean</code>	当设置为真时，提交表单的行为就不会被模拟为“单击”提交

### ➤ 向回调函数传递额外的数据

请求对象被实例化时就需要指定一个回调函数，当请求成功发送并返回响应对象后，回调方法就会被调用，在回调方法中会将与该请求对应的响应对象作为参数传入，代码如下所示。

```
def parse_page1(self, response):
    return scrapy.Request("http://www.example.com/some_page.html",
                           callback=self.parse_page2)

def parse_page2(self, response):
    # this would log http://www.example.com/some_page.html
    self.log("Visited %s" % response.url)
```

在某些情况下，可能会将发起请求时所产生的一个变量传递到回调方法中，此时可以将

`Request.meta` 属性当作上下文来使用，可以将变量存于其中，在回调方法中将回传的响应对象重新取出，具体代码示例如下：

```
def parse_page1(self, response):
    item = MyItem()
    item['main_url'] = response.url
    request = scrapy.Request("http://www.example.com/some_page.html",
                             callback=self.parse_page2)
    request.meta['item'] = item
    return request

def parse_page2(self, response):
    item = response.meta['item']
    item['other_url'] = response.url
    return item
```

#### ➤ `Request.meta` 属性的特殊键

`Request.meta` 属性除了可以存储任意的数据，Scrapy 也提供了部分的保留键用于对 request 进行功能扩展，这些键是具有特殊意义的，它们是：

- `dont_redirect`——是否遇到 304 重定向响应时不执行；
- `dont_retry`——是否不要执行错误重试；
- `handle_httpstatus_list`——处理 HTTP 状态码列表；
- `dont_merge_cookies`——是否不要合并 Cookies；
- `cookiejar`——存储 HTTP Cookie。可以表明要把 Cookie 传递下去，还可以对 Cookie 进行标记。一个 Cookie 表示一个会话 (session)，如果需要经多个会话对某网站进行爬取，则可以对 Cookie 进行标记，这样 Scrapy 就维持了多个会话。
- `redirect_urls`——重定向 URL 列表；
- `bindaddress`——用指定的 IP 地址作为请求地址。

#### ➤ Cookies

我们可以在实例化 request 对象时用一个字典对象来实现该请求中附带的 Cookie 信息，具体代码如下所示。

```
request_with_cookies = Request(url="http://www.example.com",
                               cookies={'currency': 'USD', 'country': 'UY'})
```



也可以用列表的方式指定多个 Cookie:

```
request_with_cookies = Request(url="http://www.example.com",
                                cookies=[{'name': 'currency',
                                           'value': 'USD',
                                           'domain': 'example.com',
                                           'path': '/currency'}])
```

后一种形式允许自定义 Cookie 的域和路径属性,这只有在 Cookie 被保存用于以后的请求时才有效。

某个站点返回的 Cookie 会保存在该域所指定的位置上,在下次向该站点重新发起请求时再次使用,这是属于浏览器的典型行为。如果出于某种原因要避免新的 Cookie 与已保存的 Cookie 自动合并,则可以通过设置 `Request.meta.dont_merge_cookies=True` 来指示 Scrapy 这样做。具体代码如下所示。

```
request_with_cookies = Request(url="http://www.example.com",
                                cookies={'currency': 'USD', 'country': 'UY'},
                                meta={'dont_merge_cookies': True})
```

## 4.3.3 表单处理

用 GET 方法向百度发出一个请求时,会得到一个网页搜索的表单,如果改用 POST 方法向 `http://www.baidu.com` 重新发起请求,则百度会返回一个带有与输入关键词匹配的搜索结果的响应内容,代码如下所示。

```
import requests
params = {'terms': '爬虫'}
response = requests.post('http://www.baidu.com', data = params)
print response.text
```

如果仅阅读 HTTP 协议,则是无法发现 GET 方法与 POST 方法在代码实现上的差别的,POST 和 PUT 两种 HTTP 方法都可以向服务器发送其他数据参数,更准确地说就是“表单”。

向服务器发送表单必须采用 POST 或 PUT 方法。

用 Python 在表单数据中写入请求是一件很简单的事,只要将表单数据装载到一个字典中,

然后作为 data 参数传入 post 方法或者 put 方法即可，正如上文代码所示。

爬虫在什么情况下需要处理表单？最常见的情况有以下几种：

- 网页查询；
- 用户登录/注册；
- 自动评论。

网页查询与上文的百度例子是相仿的，除了搜索引擎，不少网站都提供快速的关键词查询，以方便用户搜索信息。用 urllib 和 requests 库进行演示只是为了可以得到一个最小化原理的框架，并且能独立运行以上代码片段，这其实也是 Python 吸引人的一个地方。

### 示例：用户登录

用户登录是爬虫系统经常遇到的一种表单处理，对于网站中某些只对注册用户开放的内容，通常都要求登录，而且大多会先进行**重定向处理**，重定向在下一节讲述 HTTP 响应时再进行全面的讲解。先假定现在爬虫已被重定向到一个登录页面，要由爬虫自动进行注册或者登录。

以下是一个简化（删除所有与视觉相关的元素）后的登录页面代码：

```
<form method="post"
      action="login.php">
  <input name="user_name" />
  <input name="password" type="password" />
</form>
```

现在使用上一节中提及的 FormRequest 对象来生成一个表单请求并提交至网站。表单中的数据称为表单域（Fields），FormRequest 对象在构造时向 formdata 参数传入一个以“键-值”对形式的数据字典，具体代码如下所示。

```
return [FormRequest(url="http://www.example.com/post/action",
                    formdata={'name': 'John Doe', 'age': '27'},
                    callback=self.after_post)]
```

使用 FormRequest.from\_response() 方法模拟用户登录：通常网站通过 <input type="hidden"> 实现对某些表单字段（比如数据或登录界面中的认证令牌等）的预填充。使用 Scrapy 抓取网页时，如果想预填充或重写像用户名、用户密码等表单字段，可以使用 FormRequest.from\_response() 方法实现。下面是使用这种方法的爬虫例子：

```
from scrapy import Spider, FormRequest
```





```
class LoginSpider(Spider):
    name = 'example.com'
    start_urls = ['http://www.example.com/users/login.php']

    def parse(self, response):
        return FormRequest.from_response(
            response,
            formdata={'username': 'john', 'password': 'secret'},
            callback=self.after_login
        )

    def after_login(self, response):
        # check login succeed before going on
        if "authentication failed" in response.body:
            self.log("Login failed", level=scrapy.log.ERROR)
            return

        # continue scraping with authenticated session...
```

除了在蜘蛛中集成自动登录功能，还可以用下载器中间件（DownloaderMiddleWare）封装这一功能，通过配置就可以将其与其他蜘蛛共同使用。

### 4.3.4 下载器中间件

Scrapy 官方定义：

下载器中间件是介于下载器(Downloader)与 Scrapy 引擎(Scrapy Engine)之间的 request/response 处理的钩子框架，用于全局修改 Scrapy request 和 response 的一个轻量、底层的系统。

有时某些代码在蜘蛛执行之前要先执行，或者要对 request 进行重新调整，此时就可以使用下载器中间件这种插件系统在不修改代码的前提下直接将新的功能模块接入 Scrapy 框架中。

#### 自定义下载器中间件

编写下载器中间件十分简单。每个中间件组件定义了一个或多个方法的 Python 类，以下是自定义下载器中间件的示例模板：



```

class MyDownloaderMiddleWare(object):

    def process_request(self,request, spider):
        """
        当每个 request 通过下载中间件时，该方法被调用
        """
        pass

    def process_response(self,request, response, spider):
        """
        当完成对 request 的下载并产生 response 对象时在调用 Spider 上的 parse 方法之
前被调用
        """
        pass

    def process_exception(self,request, exception, spider):
        """
        当下载处理器 (download handler) 或 process_request() 抛出异常 (包括
IgnoreRequest 异常) 时被调用
        """
        Pass

```

下载器中间件一共提供了三个方法，从方法名就知道它们是向远程主机发送请求这一生命周期内的三个方法，它们的调用是有序与条件的，因为我们可以将其理解为下载器中间件中的事件处理方法。

以下是这三个方法返回值的说明。

#### ➤ process\_request()

必须返回其中之一：

- 如果其返回 None，则 Scrapy 继续处理该 request，执行其他中间件的相应方法，直到合适的下载器处理函数(download handler)被调用，该 request 被执行(其 response 被下载)。
- 如果其返回 response 对象，则 Scrapy 不会调用任何其他 process\_request() 或 process\_exception() 方法，或相应的下载函数；其将返回该 response。已安装的中间件的 process\_response() 方法则会在每个 response 返回时被调用。
- 如果其返回 request 对象，则 Scrapy 停止调用 process\_request 方法并重新调度返





回的 `request`。当新返回的 `request` 被执行后，相应的中间件链会根据下载的 `response` 被调用。

- 如果其 `raise` 一个 `IgnoreRequest` 异常，则安装的下载中间件的 `process_exception()` 方法会被调用。如果没有任何一个方法处理该异常，则 `request` 的 `errback(Request.errback)` 方法会被调用。如果没有代码处理抛出的异常，则该异常被忽略且不记录（不同于其他异常）。

#### ➤ `process_response()`

必须返回以下之一：

- 如果其返回一个 `response`（可以与传入的 `response` 相同，也可以是全新的对象），该 `response` 会被链中的其他中间件的 `process_response()` 方法处理。
- 如果其返回一个 `request` 对象，则中间件链停止，返回的 `request` 会被重新调度下载。处理类似于 `process_request()` 返回 `request` 所做的那样。
- 如果其抛出 `IgnoreRequest` 异常，则调用 `request` 的 `errback(Request.errback)`。如果没有代码处理抛出的异常，则该异常被忽略且不记录（不同于其他异常）。

#### ➤ `process_exception()`

应该返回以下之一：

- 如果其返回 `None`，则 `Scrapy` 继续处理该异常，接着调用已安装的其他中间件的 `process_exception()` 方法，直到所有中间件都被调用完毕，则调用默认的异常处理。
- 如果其返回一个 `response` 对象，则已安装的中间件链的 `process_response()` 方法被调用。`Scrapy` 将不会调用任何其他中间件的 `process_exception()` 方法。
- 如果其返回一个 `Request` 对象，则返回的 `request` 会被重新调用下载。这将停止中间件的 `process_exception()` 方法执行，就如返回一个 `response` 一样。

### 用于登录的下载器中间件

接下来做一个双向示例。我会采用 `Flask` 编写一个简单的服务端，这个服务端只具有一个 `login` 入口，当输入的用户名与密码正确时就会返回一个欢迎页面，反之则会返回一个包含“登录失败”字符串的响应。

此服务端例子只需要安装 `Flask`：\$ `pip install flask`。

```
# coding:utf-8
```



```

from flask import Flask,request

app = Flask(__name__)

@app.route('/login',methods=['POST'])
def login():
    print 'user login' # 当接收到请求时打印信息到控制台
    if request.form['username'] == 'ray' and request.form['password']==
'123456':
        return """<html>
<head></head>
<body>
    <h1>欢迎</h1>
    <p>Login success</p>
</body>
</html>"""
    else:
        return u"登录失败"

```

将以上代码保存到 simple\_web.py 中，然后直接运行：

```

$ python simple_web.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

```

现在服务端已运行于本地的 5000 端口上，接下来在爬虫端采用一个下载器中间件实现这个登录功能，该中间件会在蜘蛛启动前自动启动并先执行登录。

```

class LoginMiddleWare(object):

    def process_request(self,request, spider):
        # 采用 FormRequest 发出 POST 请求到上文的服务器地址
        return FormRequest(url="http://localhost:5000/login",
            formdata={'username': 'ray', 'password': '123456'})

    def process_response(self,request, response, spider):
        if u"登录失败" in response.body:
            return IgnoreRequest()

```





```
def process_exception(self, request, exception, spider):
    self.log(u"登录失败", level=scrapy.log.ERROR)
```

逻辑非常简单,但在实际运行的情况下,服务器登录成功会向客户端写入 Cookie。为了让程序更简单易懂,此处跳过了这一步,但这并不会影响爬虫的运行,因为 Scrapy 的默认运行配置中会自动启动 CookiesMiddleware 中间件以维持服务器发放的 Cookie,并在下次发出服务器请求时将 Cookie 重新写入请求对象中。

另外,我们可以将上述 LoginMiddleware 中的用户名与密码放到配置文件中,首先在配置文件中加入两个键值对:

```
# settings.py
LOGIN_USER = 'ray'
LOGIN_PWD = '123456'
LOGIN_URL = 'http://www.example.com/post/action'
```

然后定义类方法下载器中间件的 from\_crawler,在这个类方法内我们可以通过 crawler 参数获取配置值,然后动态重写下载器中间件的构造函数,使配置对象 settings 可以作为初始参数传入:

```
class LoginMiddleWare(object):

    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler.settings)

    def __init__(self, settings):
        # 此处从 settings 实例中获取具体的配置值
        self.username = settings.get('USER_NAME')
        self.password = settings.get('PASSWORD')
        self.login_url = settings.get('login_url')

    def process_request(self, request, spider):
        return FormRequest(url=self.login_url,
                           formdata={'username': self.username, 'password': self.password})

# 省略
```



## 启用下载器中间件

要启用以上的中间件我们就需要在配置文件 (settings.py) 中激活它，以下是一个例子：

```
DOWNLOADER_MIDDLEWARES = {
    'formrequest_example.middlewares.LoginMiddleWare': 543,
}
```

DOWNLOADER\_MIDDLEWARES 设置会与 Scrapy 定义的 DOWNLOADER\_MIDDLEWARES\_BASE 设置合并（但不是覆盖），而后根据顺序（order）进行排序，最后得到启用中间件的有序列表：第一个中间件是最靠近引擎的，最后一个中间件是最靠近下载器的。

关于如何分配中间件的顺序请查看 DOWNLOADER\_MIDDLEWARES\_BASE 设置，而后根据想要放置中间件的位置选择一个值。由于每个中间件执行不同的动作，中间件可能会依赖于之前（或者之后）执行的中间件，因此顺序是很重要的。

如果想禁止内置的（在 DOWNLOADER\_MIDDLEWARES\_BASE 中设置并默认启用的）中间件，则必须在项目的 DOWNLOADER\_MIDDLEWARES 设置中定义该中间件，并将其值赋为 None。例如，关闭 user-agent 中间件：

```
DOWNLOADER_MIDDLEWARES = {
    'formrequest_example.middlewares.LoginMiddleWare': 543,
    'scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware': None,
}
```

修改配置文件 settings.py，这个下载器中间件就可以宣告完成了，当启动 Scrapy 后就可以在控制台中看到输出结果。这个下载器中间件与上一节中讲述的 LoginSpider 就有着不同的意义。

蜘蛛中写入的登录逻辑只能在该蜘蛛中使用，而下载器中间件却可以将蜘蛛中的通用逻辑提出来，通过配置文件应用到所有需要进行登录的蜘蛛上。这在设计原则中称为“组合”，组合比继承更灵活，因为它可以将类之间的依赖性完全地分离到外部。这样使得代码更加清晰，而且可以在最大程度上将局部的逻辑进行重用。

## 附：Scrapy内置下载器中间件

### ► CookiesMiddleware

```
class scrapy.contrib.downloadermiddleware.cookies.CookiesMiddleware
```

该中间件使得爬取需要 Cookie（例如，使用 session）的网站成为了可能。其追踪了 Web server





发送的 Cookie，并在之后的 request 中发送回去，就如同浏览器所做的那样。

以下设置可以用来配置 Cookie 中间件：

- **COOKIES\_ENABLED**——是否启用 Cookies middleware。如果关闭，则 Cookies 将不会发送给 Web server。默认：True。
- **COOKIES\_DEBUG**——如果启用，则 Scrapy 将记录所有 request (Cookie 请求头) 发送的 Cookies 及 response 接收到的 Cookies (Set-Cookie 接收头)。默认：False。

Scrapy 通过使用 Cookiejar Request meta key 来支持单 Spider 追踪多 Cookie session。默认情况下其使用一个 cookiejar(session)，不过可以通过传递一个标示符来使用多个。

例如：

```
for i, url in enumerate(urls):
    yield scrapy.Request("http://www.example.com", meta={'cookiejar': i},
        callback=self.parse_page)
```

需要注意的是，cookiejar 键不是“黏性的 (sticky)”。需要在之后的 request 请求中接着传递。例如：

```
def parse_page(self, response):
    # do some processing
    return scrapy.Request("http://www.example.com/otherpage",
        meta={'cookiejar': response.meta['cookiejar']},
        callback=self.parse_other_page)
```

下面是启用 COOKIES\_DEBUG 的记录样例：

```
2011-04-06 14:35:10-0300 [diningcity] INFO: Spider opened
2011-04-06 14:35:10-0300 [diningcity] DEBUG: Sending cookies to: <GET
http://www.diningcity.com/netherlands/index.html>
    Cookie: clientlanguage_nl=en_EN
2011-04-06 14:35:14-0300 [diningcity] DEBUG: Received cookies from: <200
http://www.diningcity.com/netherlands/index.html>
    Set-Cookie: JSESSIONID=B~FA4DC0C496C8762AE4F1A620EAB34F38; Path=/
    Set-Cookie: ip_isocode=US
    Set-Cookie: clientlanguage_nl=en_EN; Expires=Thu, 07-Apr-2011
21:21:34 GMT; Path=/
2011-04-06 14:49:50-0300 [diningcity] DEBUG: Crawled (200) <GET
```



```
http://www.diningcity.com/netherlands/index.html> (referer: None)
[...]
```

#### ➤ DefaultHeadersMiddleware

```
class scrapy.contrib.downloadermiddleware.defaultheaders.
DefaultHeadersMiddleware
```

该中间件设置 DEFAULT\_REQUEST\_HEADERS 指定的默认 request header。

#### ➤ DownloadTimeoutMiddleware

```
class scrapy.contrib.downloadermiddleware.downloadtimeout.
DownloadTimeoutMiddleware
```

该中间件设置 DOWNLOAD\_TIMEOUT 指定的 request 下载超时时间。

#### ➤ HttpAuthMiddleware

```
class scrapy.contrib.downloadermiddleware.httpauth.HttpAuthMiddleware
```

该中间件完成某些使用 Basic access authentication（或者叫 HTTP 认证）的 Spider 生成的请求的认证过程。

在 Spider 中启用 HTTP 认证，请设置 Spider 的 http\_user 和 http\_pass 属性。

样例：

```
from scrapy.contrib.spiders import CrawlSpider
```

```
class SomeIntranetSiteSpider(CrawlSpider):
```

```
    http_user = 'someuser'
    http_pass = 'somepass'
    name = 'intranet.example.com'
```

```
    # .. rest of the spider code omitted ...
```

#### ➤ HttpCacheMiddleware

```
class scrapy.contrib.downloadermiddleware.httpcache.HttpCacheMiddleware
```

该中间件为所有 HTTP request 及 response 提供了底层（low-level）缓存支持。其由 cache 存储后端及 cache 策略组成。





Scrapy 提供了两种 HTTP 缓存存储后端:

- Filesystem storage backend (默认值);
- DBM storage backend。

可以使用 HTTPCACHE\_STORAGE 设定来修改 HTTP 缓存存储后端, 也可以实现自己的存储后端。

Scrapy 提供了两种缓存策略:

- RFC2616 策略;
- Dummy 策略 (默认值)。

可以使用 HTTPCACHE\_POLICY 设定来修改 HTTP 缓存存储后端, 也可以实现自己的存储策略。

### Dummy 策略 (默认值)

该策略不考虑任何 HTTP Cache-Control 指令。每个 request 及其对应的 response 都被缓存。当相同的 request 发生时, 其不发送任何数据, 直接返回 response。

Dummpy 策略对于测试 Spider 十分有用, 能使 Spider 运行更快(不需要每次等待下载完成), 同时在没有网络连接时也能测试。目的是为了能够回放 Spider 的运行过程, 使之与之前的运行过程一模一样。

使用这个策略, 设置 HTTPCACHE\_POLICY 为 scrapy.contrib.httpcache.DummyPolicy。

### RFC2616 策略

该策略提供了符合 RFC2616 的 HTTP 缓存, 例如, 符合 HTTP Cache-Control, 针对生产环境并且应用在持续性运行环境。该策略能避免下载未修改的数据 (用来节省带宽, 提高爬取速度)。

实现了:

- 当 no-store cache-control 指令设置时不存储 response/request。
- 当 no-cache cache-control 指定设置时不从 cache 中提取 response, 即使 response 为最新。
- 根据 max-age cache-control 指令来计算保存时间 (freshness lifetime)。
- 根据 Expires 指令来计算保存时间 (freshness lifetime)。
- 根据 response 包头的 Last-Modified 指令来计算保存时间 (freshness lifetime, Firefox 使用的启发式算法)。
- 根据 response 包头的 age 计算当前年龄 (current age)。
- 根据 Date 计算当前年龄 (current age)。



- 根据 response 包头的 Last-Modified 验证老旧的 response。
- 根据 response 包头的 ETag 验证老旧的 response。

使用这个策略, 设置 `HTTPCACHE_POLICY` 为 `scrapy.contrib.httpcache.RFC2616Policy`。

### Filesystem storage backend (默认值)

文件系统存储后端可以用于 HTTP 缓存中间件。

使用该存储端, 设置 `HTTPCACHE_STORAGE` 为 `scrapy.contrib.httpcache.FilesystemCacheStorage`

每个 request/response 组存储在不同的目录中, 包含下列文件。

- `request_body`: 请求对象的原始正文;
- `request_headers`: 请求头对象;
- `response_body`: 响应对象的原始正文;
- `response_headers`: 响应头对象;
- `meta`: 以 Python repr() 格式 (grep-friendly 格式) 存储的该缓存资源的一些元数据。
- `pickled_meta`: 与 meta 相同的元数据, 不过使用 pickle 来获得更高效的反序列化性能。

目录的名称与 request 的指纹 (`scrapy.utils.request.fingerprint`) 有关, 而二级目录是为了避免在同一文件夹下有太多文件 (这在很多文件系统中是十分低效的)。目录的例子:

```
/path/to/cache/dir/example.com/72/72811f648e718090f041317756c03adb0ada46c7
```

### DBM 存储后端

同时有 DBM 存储后端可以用于 HTTP 缓存中间件。默认情况下, 其采用 `anydbm` 模块, 也可以通过 `HTTPCACHE_DBM_MODULE` 设置进行修改。

使用该存储端, 设置 `HTTPCACHE_STORAGE` 为 `scrapy.contrib.httpcache.DbmCacheStorage`

### LevelDB 存储后端

LevelDB 存储后端也可用于 HTTP 缓存中间件。

因为只有一个进程可以同时访问 LevelDB 数据库, 所以不推荐将此后端用于开发。因此, 不能同时为同一个蜘蛛运行爬网并打开 Scrapy shell。

按以下方式可启用 LevelDB 作为存储后端:

- (1) 设置 `HTTPCACHE_STORAGE` 为 `scrapy.contrib.httpcache.LevelDbCacheStorage`;
- (2) 使用 `$ pip install leveldb` 安装 LevelDB 的程序包。





## HTTPCache 中间件设置

HttpCacheMiddleware 可以通过以下设置进行配置，如下表所示。

配 置 项	默 认 值	说 明
HTTPCACHE_ENABLED	False	HTTP 缓存是否开启
HTTPCACHE_EXPIRATION_SECS	0	缓存的 request 的超时时间，单位秒。超过这个时间的缓存 request 将会被重新下载。如果为 0，则缓存的 request 将永远不会超时
HTTPCACHE_DIR	'httpcache'	存储（底层的）HTTP 缓存的目录。如果为空，则 HTTP 缓存会被关闭。如果为相对目录，则相对于项目数据目录（project data dir）。更多内容请参考默认的 Scrapy 项目结构
HTTPCACHE_IGNORE_HTTP_CODES	[]	不缓存设置中的 HTTP 返回值（code）的 request
HTTPCACHE_IGNORE_MISSING	False	如果启用，则在缓存中没找到的 request 会被忽略，不下载
HTTPCACHE_IGNORE_SCHEMES	['file']	不缓存这些 URI 标准(scheme) 的 response
HTTPCACHE_STORAGE	'scrapy.contrib.httpcache.FilesystemCacheStorage'	实现缓存存储后端的类
HTTPCACHE_DBM_MODULE	'anydbm'	DBM 存储后端的数据库模块。该设定针对 DBM 后端
HTTPCACHE_POLICY	scrapy.contrib.httpcache.DummyPolicy	实现缓存策略的类

### ➤ HttpCompressionMiddleware

```
class scrapy.contrib.downloadermiddleware.httpcompression.  
HttpCompressionMiddleware
```

该中间件提供了对压缩（gzip,deflate）数据的支持。

- **COMPRESSION\_ENABLED**——压缩中间件是否开启。默认为 True。

### ➤ ChunkedTransferMiddleware

```
class scrapy.contrib.downloadermiddleware.chunked.  
ChunkedTransferMiddleware
```

该中间件添加了对分块传输编码的支持。

### ➤ HttpProxyMiddleware

```
class scrapy.contrib.downloadermiddleware.httpproxy.HttpProxyMiddleware
```

该中间件提供了对 request 设置 HTTP 代理的支持。可以通过在 Request 对象中设置 proxy 元数据来开启代理。

类似于 Python 标准库模块 urllib 及 urllib2，其使用了下列环境变量：

- http\_proxy
- https\_proxy
- no\_proxy

### ➤ RedirectMiddleware

```
class scrapy.contrib.downloadermiddleware.redirect.RedirectMiddleware
```

该中间件根据 response 的状态处理重定向的 request。被重定向的 request 的 URL 可以通过 Request.meta 的 redirect\_urls 键找到。

RedirectMiddleware 可以通过下列设置进行配置：

- REDIRECT\_ENABLED——是否启用 Redirect 中间件。默认为 True
- REDIRECT\_MAX\_TIMES——单个 request 被重定向的最大次数。默认为 20。

如果 Request.meta 包含 dont\_redirect 键，则该 request 会被此中间件忽略。

### ➤ MetaRefreshMiddleware

```
class scrapy.contrib.downloadermiddleware.redirect.MetaRefreshMiddleware
```

该中间件根据 meta-refresh html 标签处理 request 重定向。

MetaRefreshMiddleware 可以通过以下设定进行配置：

- METAREFRESH\_ENABLED——Meta Refresh 中间件是否启用。默认为 True。
- METAREFRESH\_MAXDELAY——Meta refresh 重定向的最大延迟。



- REDIRECT\_MAX\_METAREFRESH\_DELAY——跟进重定向的最大 meta-refresh 延迟（单位：秒）。默认为 100。

该中间件遵循 RedirectMiddleware 描述的 REDIRECT\_MAX\_TIMES 设定, dont\_redirect 及 redirect\_urls。

#### ➤ RetryMiddleware

```
class scrapy.contrib.downloadermiddleware.retry.RetryMiddleware
```

该中间件将重试可能由于临时的问题，例如，连接超时或者 HTTP 500 错误导致失败的页面。

爬取进程会收集失败的页面，Spider 爬取完所有正常（不失败）的页面后重新调度。一旦没有更多需要重试的失败页面，该中间件将发送一个信号（retry\_complete），其他插件可以监听该信号。

RetryMiddleware 可以通过下列设定进行配置：

- RETRY\_ENABLED——Retry Middleware 是否启用。默认为 True。
- RETRY\_TIMES——包括第一次下载，最多的重试次数。默认为 2。
- RETRY\_HTTP\_CODES——重试的 response 返回值（code）。其他错误（DNS 查找问题、连接失败及其他）则一定会进行重试。默认为[500, 502, 503, 504, 400, 408]。

关于 HTTP 错误的考虑：根据 HTTP 协议，可能想在设定 RETRY\_HTTP\_CODES 中移除 400 错误。该错误被默认为这个代码经常被用来指示服务器过载（overload）。而在这种情况下，我们想进行重试。

如果 Request.meta 包含 dont\_retry 键，则该 request 会被此中间件忽略。

#### ➤ RobotsTxtMiddleware

```
class scrapy.contrib.downloadermiddleware.robotstxt.RobotsTxtMiddleware
```

该中间件过滤所有 robots.txt exclusion standard 中禁止的 request。

确认该中间件及 ROBOTSTXT\_OBEY 设置被启用以确保 Scrapy 遵守 robots.txt 中的约定。

**注意：**如果在一个网站中使用了多个并发请求，则 Scrapy 仍然可能下载一些被禁止的页面。这是由于这些页面是在 robots.txt 被下载前被请求的。这是当前 robots.txt 中间件已知的限制，并将在未来进行修复。

### ➤ DownloaderStats

```
class scrapy.contrib.downloadermiddleware.stats.DownloaderStats
```

保存所有通过的 request、response 及 exception 的中间件。

必须启用 DOWNLOADER\_STATS 来启用该中间件。

### ➤ UserAgentMiddleware

```
class scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware
```

用于覆盖 Spider 的默认 user agent 的中间件。

要使 Spider 能覆盖默认的用户 agent，其 user\_agent 属性必须被设置。

### ➤ AjaxCrawlMiddleware

```
class scrapy.contrib.downloadermiddleware.ajaxcrawl.AjaxCrawlMiddleware
```

根据 meta-fragment html 标签查找“AJAX 可爬取”页面的中间件。查看 <https://developers.google.com/webmasters/ajax-crawling/docs/getting-started> 来获得更多内容。

注解：即使没有启用该中间件，Scrapy 仍能查找类似 `http://example.com/!#foo=bar` 的“AJAX 可爬取”页面。AjaxCrawlMiddleware 针对的是不具有“!”的 URL，通常发生在“index”或者“main”页面中。

AJAXCRAWL\_ENABLED——启用本中间件。

## 4.4 处理HTTP响应

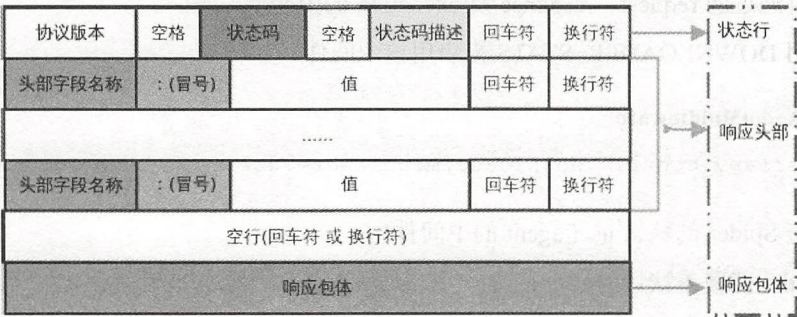
HTTP 响应 (HttpResponse) 的处理重点在于对响应数据的分析与提取。爬虫系统所采集的数据可谓千奇百怪，除了普通网页，可能还需要处理图形图像、纯文本文件，甚至是其他半结构化或非结构化数据。

对于普通的网页，本章会再度深入到选择器，并且用更为高效的 XPath 来分析、定位和提取网页正文中我们真正需要采集的数据。而对于那些结构化和非结构化的数据，就需要我们能充分地使用别人造好的轮子来解决问题，这也正是 Python 的一大长处（强大的 Python 社区几乎没有我们找不到的轮子）。最后，我们回归问题的本质，看一下如何借助一些高效的工具来帮助我们分析 HTTP 响应的内容。



### 4.4.1 HTTP响应

如下图所示，HTTP 响应的格式与请求的格式十分类似。



HTTP 响应的协议格式代码如下：

```
<status-line>
<headers>
<blank line>
[<response-body>]
```

下面对响应报文格式进行简单的分析。

状态行：状态行由 HTTP 协议版本字段、状态码和状态码的描述文本三个部分组成，它们之间使用空格隔开。

状态码由三位数字组成，第一位数字表示响应的类型，常用的状态码有五大类。

- 1xx：表示服务器已接收了客户端请求，客户端可继续发送请求。
- 2xx：表示服务器已成功接收到请求并进行处理。
- 3xx：表示服务器要求客户端重定向。
- 4xx：表示客户端的请求有非法内容。
- 5xx：表示服务器未能正常处理客户端的请求而出现意外错误。

状态码描述文本有如下取值。

- 200 OK：表示客户端请求成功；
- 400 Bad Request：表示客户端请求有语法错误，不能被服务器所理解；
- 401 Unauthorized：表示请求未经授权，该状态代码必须与 WWW-Authenticate 报头域一起使用；

- 403 Forbidden: 表示服务器收到请求, 但是拒绝提供服务, 通常会在响应正文中给出不提供服务的原因;
- 404 Not Found: 请求的资源不存在, 例如, 输入了错误的 URL;
- 500 Internal Server Error: 表示服务器发生不可预期的错误, 导致无法完成客户端的请求;
- 503 Service Unavailable: 表示服务器当前不能够处理客户端的请求, 在一段时间之后, 服务器可能会恢复正常。

响应头部: 响应头可能包括以下内容。

- Location: Location 响应报头域用于重定向接收者到一个新的位置。例如, 客户端请求的页面已不在原先的位置, 为了让客户端重定向到这个页面新的位置, 服务器端可以发回 Location 响应报头后使用重定向语句, 让客户端去访问新的域名所对应的服务器上的资源。
- Server: Server 响应报头域包含了服务器用来处理请求的软件信息及其版本。它和 User-Agent 请求报头域是相对应的, 前者发送服务器端软件的信息, 后者发送客户端软件 (浏览器) 和操作系统的信息。
- Vary: 指示不可缓存的请求头列表。
- Connection: 连接方式。对于请求来说: close (告诉 Web 服务器或者代理服务器, 在完成本次请求的响应后断开连接, 不等待本次连接的后续请求了); keepalive (告诉 Web 服务器或者代理服务器, 在完成本次请求的响应后保持连接, 等待本次连接的后续请求)。对于响应来说: close (连接已经关闭); keepalive (连接保持, 在等待本次连接的后续请求)。
- Keep-Alive: 如果浏览器请求保持连接, 则该头部表明希望 Web 服务器保持连接多长时间 (秒)。例如, Keep-Alive: 300。
- WWW-Authenticate: WWW-Authenticate 响应报头域必须被包含在 401 (未授权的) 响应消息中, 这个报头域和前面讲到的 Authorization 请求报头域是相关的, 当客户端收到 401 响应消息后, 就要决定是否请求服务器对其进行验证。如果要求服务器对其进行验证, 就可以发送一个包含了 Authorization 报头域的请求。空行: 最后一个响应头部之后是一个空行, 发送回车符和换行符, 通知服务器以下不再有响应头部。响应主体: 服务器返回给客户端的文本信息、

在响应中唯一真正的区别在于第一行中用状态信息代替了请求信息。状态行 (status line) 通过提供一个状态码来说明所请求的资源情况。以下就是一个 HTTP 响应的例子:



```
HTTP/1.1 200 OK
Date: Sat, 31 Dec 2005 23:59:59 GMT
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 122
<html>
<head>
<title>Homepage</title>
</head>
<body>
<!-- body goes here -->
</body>
</html>
```

在状态行之后是一些首部。通常，服务器会返回一个名为 **Data** 的首部，用来说明响应生成的日期和时间（服务器通常还会返回一些关于其自身的信息，尽管并非必需的）。接下来的两个首部大家应该很熟悉，就是与 **POST** 请求中一样的 **Content-Type** 和 **Content-Length**。在本例中，首部 **Content-Type** 指定了 **MIME** 类型 **HTML** (**text/html**)，其编码类型是 **ISO-8859-1**（这是针对美国英语资源的编码标准）。响应主体所包含的就是所请求资源的 **HTML** 源文件（尽管还可能包含纯文本或其他资源类型的二进制数据）。浏览器将把这些数据显示给用户。

注意，这里并没有指明针对该响应的请求类型，不过这对于服务器来说并不重要。客户端知道每种类型的请求将返回什么类型的数据，并决定如何使用这些数据。

HTTP响应头

常用响应头域如下表所示。

头 域	解 释	示 例
Accept-Ranges	表明服务器是否支持指定范围请求及哪种类型的分段请求	Accept-Ranges: bytes
Age	从原始服务器到代理缓存形成的估算时间（以秒计，非负）	Age: 12
Allow	对某网络资源的有效请求行为，不允许则返回 405	Allow: GET, HEAD
Cache-Control	告诉所有的缓存机制是否可以缓存及哪种类型	Cache-Control: no-cache
Content-Encoding	Web 服务器支持的返回内容压缩编码类型	Content-Encoding: gzip

续表

头 域	解 释	示 例
Content-Language	响应体的语言	Content-Language: en,zh
Content-Length	响应体的长度	Content-Length: 348
Content-Location	请求资源可替代的备用的另一地址	Content-Location: /index.htm
Content-MD5	返回资源的 MD5 校验值	Content-MD5: Q2hly2sgSW50ZWdyaXR5IQ==
Content-Range	在整个返回体中本部分的字节位置	Content-Range: bytes 21010-47021/47022
Content-Type	返回内容的 MIME 类型	Content-Type: text/html; charset=utf-8
Date	原始服务器消息发出的时间	Date: Tue, 15 Nov 2010 08:12:31 GMT
ETag	请求变量的实体标签的当前值	ETag: "737060cd8c284d8af7ad3082f2095 82d"
Expires	响应过期的日期和时间	Expires: Thu, 01 Dec 2010 16:00:00 GMT
Last-Modified	请求资源的最后修改时间	Last-Modified: Tue, 15 Nov 2010 12:45:26 GMT
Location	用来重定向接收方到非请求 URL 的位置来完成请求或标识新的资源	Location: http://www.dotnetage.com/archives/ 94.html
Pragma	包括实现特定的指令, 它可应用到响应链上的任何接收方	Pragma: no-cache
Proxy-Authenticate	它指出认证方案和可应用到代理的该 URL 上的参数	Proxy-Authenticate: Basic
refresh	应用于重定向或一个新的资源被创造, 在 5 秒之后重定向 (由网景提出, 被大部分浏览器支持)	Refresh: 5; url=http://www.dotnetage.com/archi ves/94.html
Retry-After	如果实体暂时不可取, 则通知客户端在指定时间之后再次尝试	Retry-After: 120
Server	Web 服务器软件名称	Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Set-Cookie	设置 HTTP Cookie	Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1



续表

头 域	解 释	示 例
Trailer	指出头域在分块传输编码的尾部存在	Trailer: Max-Forwards
Transfer-Encoding	文件传输编码	Transfer-Encoding:chunked
Vary	告诉下游代理是使用缓存响应还是从原始服务器请求	Vary: *
Via	告知代理客户端响应是通过哪里发送的	Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
Warning	警告实体可能存在的问题	Warning: 199 Miscellaneous warning
WWW-Authenticate	表明客户端请求实体应该使用的授权方案	WWW-Authenticate: Basic

4.4.2 Scrapy的响应对象

Scrapy 提供了一个 response 对象用于描述 HTTP 响应，并且根据其响应内容的格式衍生出 TextResponse、HtmlResponse 和 XmlResponse 对象。

response对象

```
class scrapy.http.Response(url[, status=200, headers, body, flags])
```

response 对象一般由下载器发出请求，获得服务器的响应后产生并传递至蜘蛛的 parse 方法中。其属性说明如下表所示。

属 性	类 型	说 明
url	string	响应的 URL 地址，此地址未必一定等于 request.url，被重定向后此地址就会与 request.url 有所差异
meta	dict	从 request.meta 中复制的 meta 对象信息
headers	dict	响应头对象
status	integer	HTTP 响应状态码
body	string	没有进行任何编码操作的响应正文字符串
flags	list	此属性保存了当前响应对象的初始化值的列表的一个影本。Flags 是一些用于标记 response 对象的标签，例如：cached、redirected 等。它们用于记录爬虫引擎日志
request	request	与此响应对应的请求对象实例

## TextResponse对象

```
class scrapy.http.TextResponse(url[, encoding[, ...]])
```

TextResponse 对象在 response 对象的基础上增加了文字编码能力，一般上只用于承载二进制数据，例如，图像、声音或者多媒体文件。

TextResponse 对象的构造函数比原生的 response 增加了一个 encoding 参数，这个参数是一个字符串值，用于指定当前 response 对象的文本内容采用哪种编码方式。如果将 TextResponse 以 Unicode 形式进行构造，那么该 TextResponse 实例中的内容会被自动以 Unicode 形式进行编码。如果将其设置为 None（默认值），则响应对象就会自动从响应头中寻找当前内容的具体编码方法。

### ➤ 属性

TextResponse 比原生 response 扩展了以下两个属性。

- encoding 以字符串形式返回当前响应对象中内容的编码方式。该属性会按照以下顺序获取：
  - 构造函数编码参数中传递的编码值。
  - Content-Type HTTP 头中声明的编码；如果这种编码是无效的（即未知的），则它将被忽略，并尝试下一个解析机制。
  - 响应正文中声明的编码。TextResponse 类不提供任何特殊的功能。但 HtmlResponse 和 XmlResponse 类可以。
  - 通过查看响应主体来推断编码。这是更脆弱的方法，但也是最后一种尝试。
- selector——使用响应作为目标的选择器实例。在第一次访问时，选择器是采取惰性构造的。

### ➤ 帮助方法

TextResponse 对象还提供了以下基于 response 对象的帮助方法：

- body\_as\_unicode() —— 将 body 的内容以 Unicode 的编码方式返回 response.body.decode(response.encoding)，但不等于 unicode(response.body)。后者会使用系统的默认编码方式（通常是 ASCII）将 body 内容转换为 Unicode。
- xpath(query) —— TextResponse.selector.xpath(query) 的一个捷径写法，如 response.xpath('//p')。
- css(query) —— TextResponse.selector.css(query) 的一个捷径写法，如 response.css('p')。



### HtmlResponse对象

```
class scrapy.http.HtmlResponse(url[, ...])
```

HtmlResponse 继承自 TextResponse, 它只是增加了可以自动从 HTML 的 http-equivmeta 属性中识别文本编码的功能。

### XmlResponse对象

```
class scrapy.http.XmlResponse(url[, ...])
```

XmlResponse 继承自 TextResponse, 它只是增加了可以自动从 XML 的声明标记中识别文本编码的功能。

## 4.4.3 深入选择器

当抓取网页时, 常见的任务是从 HTML 源码中提取数据。现有的一些库可以达到这个目的。

- BeautifulSoup 是非常流行的网页分析库, 它基于 HTML 代码的结构来构造一个 Python 对象, 对不良标记的处理也非常合理, 但它有一个缺点: 慢。
- lxml 是一个基于 ElementTree (不是 Python 标准库的一部分) 的 Python 化的 XML 解析库 (也可以解析 HTML)。

Scrapy 提取数据有自己的一套机制。它们被称作选择器 (selectors), 因为它们通过特定的 XPath 或者 CSS 表达式来“选择”HTML 文件中的某个部分。

XPath 是一门用来在 XML 文件中选择节点的语言, 也可以用在 HTML 上。CSS 是一门将 HTML 文档样式化的语言。选择器由它定义, 并与特定的 HTML 元素的样式相关联。

Scrapy 选择器构建于 lxml 库之上, 这意味着它们在速度和解析准确性上非常相似。

因此从本节开始, 在没有必要的情况下不再采用 BeautifulSoup 作为文档分析工具, 在我的诸多真实爬虫项目中几乎没有它的身影了, 我更喜欢使用 XPath 作为选择器对文档进行搜寻。XPath 对于 XML 结构文档的查找可称得上极速了, 唯一的缺点就是它需要先学习 XPath 和 XQuery 的相关语法, 学习曲线要比采用 BeautifulSoup 或者 CSS 选择器更陡峭。但性能与学习曲线通常来说都是成反比的, 要有更好的性能就一定得有更多的付出, 可以说这也是写出好程序的一个必备的觉悟。

同样, CSS 选择器也不会后面的讨论范畴中, 因为只要掌握了 XPath 选择器, 要学习 CSS 选择器就更加容易了 (至少它没有 XPath 那么复杂, 会写样式表的读者可以很快掌握, 原理完全一样)。

Scrapy selector 是以文字 (text) 或 TextResponse 构造的 Selector 实例。其根据输入的类型自动选择最优的分析方法 (XML vs. HTML)。为了先从认知上对 Scrapy 的选择器有一个感性的认识, 我们可以进入 Python shell 来实验一下选择器的用法:

```
>>> from scrapy.selector import Selector
>>> from scrapy.http import HtmlResponse
```

以文字构造 Selector:

```
>>> body = '<html><body><span>good</span></body></html>'
>>> Selector(text=body).xpath('//span/text()').extract()
[u'good']
```

以 response 构造 Selector:

```
>>> response = HtmlResponse(url='http://example.com', body=body)
>>> Selector(response=response).xpath('//span/text()').extract()
[u'good']
```

为了方便起见, response 对象有一个 selector 属性, 可以随时使用该快捷属性:

```
>>> response.selector.xpath('//span/text()').extract()
[u'good']
```

从以上代码中应该可以看出选择器的使用方法了:

- (1) 构造选择器 Selector 对象。
- (2) 用 xpath() 或 css() 指定选择路径表达式。
- (3) 调用 extract() 执行选择器并产生结果对象。

## 选择器的使用

上面初步了解了选择器的基本构造方法, 应该在哪里使用它呢? 当然是在蜘蛛的 parse 方法中。在前文中一再强调, 如果从 Spider 继承实现蜘蛛就必须手工实现 parse 方法, 之前的代码示例也是在 parse 方法中采用 BeautifulSoup 来从响应对象中提取数据, 现在要做的就是用 Scrapy 的原生选择器取代 BeautifulSoup。以下是具体的改写办法:

```
def parse(self, response):
    selector = Selector(response)
```



```
for node in selector.xpath('//item').extract():
    feed_item = FeedItem()
    feed_item['title'] = node.xpath('title/text()').extract_first()
    feed_item['link'] = node.xpath('link/text()').extract_first()
    feed_item['desc'] = node.xpath('desc/text()').extract_first()
    feed_item['pub_date'] = node.xpath('pub_date/text()').extract_first()
    yield feed_item
```

在代码量上似乎比 BeautifulSoup 的写法多了一些，但比起 BeautifulSoup 在大规模运行时所带来的低效，这点代码的付出绝对是值得的。这里对以上代码进行解释，首先 `selector.xpath('//item')` 的意思是从根元素下选择所有 `item` 标记的元素对象，`extract()` 方法返回的是一个选择器结果的集合。在 `for` 循环体中，`node` 代表的是单个的节点上的选择器（注意：这不是节对象而是选择器）。因此可以再次在节点上应用 XPath 表达式：`node.xpath('title/text()')`，指从当前节点下选择 `title` 元素，并对该元素执行 `text()` 函数以获取其标签内的文字。最后用 `extract_first()` 是因为这次我们要执行 XPath 返回的结果应该是一个字符串值而不再是选择器。

有了 BeautifulSoup 的示例代码作为知识铺垫，学习 Scrapy 的原生选择器是不是感觉就没有那么难懂了呢？另外，XPath 也不会是什么难啃的骨头，你并不需要学习它所有的技术细节，因为在爬虫领域，很多 XPath 的内容其实也用不上，我们只需要知道如何来选中需要的节点，如何从节点中取出值就足够了。实际上 XPath 和 Linux 的文件目录路径也是非常相似的，只不过是节点看作目录而已。

要快速而深刻地掌握一门技术，最重要的是先了解其本质的脉络，其他技术部分都只是一些辅助方案与简化用法的内容而已。

### XPath的常规用法参考

除了上述例中的用法，接下来介绍一些可能会在实战中遇到的 XPath 表达式和选择器的配合用法。假设有以下网页：

```
<html>
<head>
  <base href='http://example.com/' />
  <title>示例网页</title>
</head>
<body>
```

```

<div id='images'>
  <a href='image1.html'>图片 1<br /><img src='image1_thumb.jpg' /></a>
  <a href='image2.html'>图片 2 <br /><img src='image2_thumb.jpg' /></a>
  <a href='image3.html'>图片 3<br /><img src='image3_thumb.jpg' /></a>
  <a href='image4.html'>图片 4<br /><img src='image4_thumb.jpg' /></a>
  <a href='image5.html'>图片 5<br /><img src='image5_thumb.jpg' /></a>
</div>
</body>
</html>

```

```

2017-11-27 13:43:53 [scrapy.utils.log] INFO: Overridden settings: {'NEWSPIDER_MODULE': 'chinanews_crawler.spiders', 'FEED_URI': 'result.json', 'DUPEFILTER_CLASS': 'scrapy.dupefilters.BaseDupeFilter', 'SPIDER_MODULES': ['chinanews_crawler.spiders'], 'BOT_NAME': 'chinanews', 'LOGSTATS_INTERVAL': 0, 'FEED_FORMAT': 'json'}
2017-11-27 13:43:53 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.feedexport.FeedExporter',
 'scrapy.extensions.memusage.MemoryUsage',
 'scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.corestats.CoreStats']
2017-11-27 13:43:53 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2017-11-27 13:43:53 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.spidermiddlewares.referrer.ReferrerMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2017-11-27 13:43:53 [scrapy.middleware] INFO: Enabled item pipelines:
[]
2017-11-27 13:43:53 [scrapy.extensions.telnet] DEBUG: Telnet console listening on 127.0.0.1:6023
2017-11-27 13:43:53 [scrapy.core.engine] INFO: Spider opened
2017-11-27 13:43:53 [scrapy.downloadermiddlewares.redirect] DEBUG: Redirecting (301) to <GET https://doc.scrapy.org/en/latest/_static/selectors-sample1.html> from <GET http://doc.scrapy.org/en/latest/_static/selectors-sample1.html>
2017-11-27 13:43:53 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://doc.scrapy.org/en/latest/_static/selectors-sample1.html> (referer: None)
[s] Available Scrapy objects:
[s] scrapy scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s] crawler <scrapy.crawler.Crawler object at 0x10b217ad0>
[s] item {}
[s] request <GET http://doc.scrapy.org/en/latest/_static/selectors-sample1.html>
[s] response <200 https://doc.scrapy.org/en/latest/_static/selectors-sample1.html>
[s] settings <scrapy.settings.Settings object at 0x10b217850>
[s] spider <DefaultSpider 'default' at 0x10b542800>
[s] Useful shortcuts:
[s] fetch(url[, redirect=True]) Fetch URL and update local objects (by default, redirects are followed)
[s] fetch(req) Fetch a scrapy.Request and update local objects
[s] shell() Shell help (print this help)
[s] view(response) View response in a browser
>>>

```

首先，打开 shell：

```
$ scrapy shell http://doc.scrapy.org/en/latest/_static/selectors-sample1.html
```

接着，当 shell 载入后，将获得名为 response 的 shell 变量，其为响应的 response，并且在 response.selector 属性上绑定了一个 selector。

此时我们就可以在命令行窗口直接键入代码来查看效果了。

因为处理的是 HTML，选择器将自动使用 HTML 语法分析。通过查看 HTML code 页面的源码，我们构建一个 XPath 来选择 title 标签内的文字：



```
>>> response.selector.xpath('//title/text()')
[<Selector (text) xpath=//title/text()>]
```

由于在 response 中使用 XPath、CSS 查询十分普遍, 因此, Scrapy 提供了两个实用的快捷方式: response.xpath() 和 response.css()。

```
>>> response.xpath('//title/text()')
[<Selector (text) xpath=//title/text()>]
>>> response.css('title::text')
[<Selector (text) xpath=//title/text()>]
```

.xpath() 和 .css() 方法返回一个 SelectorList 类的实例, 它是一个新选择器的列表。这个 API 可以用来快速提取嵌套数据。

为了提取真实的原文数据, 需要调用 .extract() 方法, 代码如下:

```
>>> response.xpath('//title/text()').extract()
[u'Example website']
```

注意, CSS 选择器可以使用 CSS3 伪元素 (pseudo-elements) 来选择文字或者属性节点:

```
>>> response.css('title::text').extract()
[u'Example website']
```

现在我们将得到根 URL (base URL) 和一些图片链接:

```
>>> response.xpath('//base/@href').extract()
[u'http://example.com/']
```

```
>>> response.css('base::attr(href)').extract()
[u'http://example.com/']
```

```
>>> response.xpath('//a[contains(@href, "image")]/@href').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']
```

```
>>> response.css('a[href*=image]::attr(href)').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']

>>> response.xpath('//a[contains(@href, "image")]/img/@src').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']

>>> response.css('a[href*=image] img::attr(src)').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']
```

### ► 嵌套选择器(selectors)

选择器方法(.xpath()或.css())返回相同类型的选择器列表,因此也可以对这些选择器调用选择器方法。下面是一个例子:

```
>>> links = response.xpath('//a[contains(@href, "image")]')
>>> links.extract()
[u'<a href="image1.html">Name: My image 1 <br></a>',
 u'<a href="image2.html">Name: My image 2 <br></a>',
 u'<a href="image3.html">Name: My image 3 <br></a>',
 u'<a href="image4.html">Name: My image 4 <br></a>',
 u'<a href="image5.html">Name: My image 5 <br></a>']

>>> for index, link in enumerate(links):
    args = (index, link.xpath('@href').extract(), link.xpath('img/@src').extract())
    print 'Link number %d points to url %s and image %s' % args
```



```
Link number 0 points to url [u'image1.html'] and image [u'image1_thumb.jpg']
Link number 1 points to url [u'image2.html'] and image [u'image2_thumb.jpg']
Link number 2 points to url [u'image3.html'] and image [u'image3_thumb.jpg']
Link number 3 points to url [u'image4.html'] and image [u'image4_thumb.jpg']
Link number 4 points to url [u'image5.html'] and image [u'image5_thumb.jpg']
```

### ► 结合正则表达式使用选择器 (selectors)

Selector 也有一个 `.re()` 方法, 用来通过正则表达式提取数据。然而, 不同于使用 `.xpath()` 或者 `.css()` 方法, `.re()` 方法返回 Unicode 字符串的列表, 所以无法构造嵌套式的 `.re()` 调用。

下面是一个例子, 从上面的 HTML code 中提取图像名字:

```
>>> response.xpath('//a[contains(@href, "image")]/text()').re(r'Name:\s*(.*)')
[u'My image 1',
 u'My image 2',
 u'My image 3',
 u'My image 4',
 u'My image 5']
```

### ► 使用相对XPaths

如果使用嵌套的选择器, 并使用起始为 `/` 的 XPath, 那么该 XPath 将对文档使用绝对路径, 而且调用的 Selector 不是相对路径。比如, 假想提取在 `<div>` 元素中的所有 `<p>` 元素。首先, 先得到所有 `<div>` 元素:

```
>>> divs = response.xpath('//div')
```

开始时, 可能会尝试使用下面的错误方法, 因为它其实是从整篇文档而不仅仅是从那些 `<div>` 元素内部提取所有 `<p>` 元素的:

```
>>> for p in divs.xpath('//p'):
...     print p.extract()
```

下面是比较合适的处理方法 (注意, `./p` 是 XPath 的点前缀):

```
>>> for p in divs.xpath('./p'):
...     print p.extract()
```

另一种常见的情况是提取所有直系 `<p>` 的结果:

```
>>> for p in divs.xpath('p'):
...     print p.extract()
```

### ➤ 正则表达式

例如，在 XPath 的 `starts-with()` 或 `contains()` 无法满足需求时，`test()` 函数是非常有用的。

在列表中选择有 “class” 元素且结尾为一个数字的链接：

```
>>> from scrapy import Selector
>>> doc = """
... <div>
...     <ul>
...         <li class="item-0"><a href="link1.html">first item</a></li>
...         <li class="item-1"><a href="link2.html">second item</a></li>
...         <li class="item-inactive"><a href="link3.html">third item</a></li>
...         <li class="item-1"><a href="link4.html">fourth item</a></li>
...         <li class="item-0"><a href="link5.html">fifth item</a></li>
...     </ul>
... </div>
... """
>>> sel = Selector(text=doc, type="html")
>>> sel.xpath('//li//@href').extract()
[u'link1.html', u'link2.html', u'link3.html', u'link4.html', u'link5.html']
>>> sel.xpath('//li[re:test(@class, "item-\d$")]//@href').extract()
[u'link1.html', u'link2.html', u'link4.html', u'link5.html']
>>>
```

**警告：**C 语言库 `libxslt` 原生不支持 EXSLT 正则表达式，`lxml` 在实现时使用了 Python `re` 模块的钩子。因此，在 XPath 表达式中使用 `regexp` 函数可能会牺牲少量的性能。

### ➤ 集合操作

集合操作可以方便地用于在提取文字元素前从文档树中去除一部分。

例如，使用 `itemsscopes` 组和对应的 `itemprops` 来提取微数据（来自 <http://schema.org/Product> 的样本内容）：

```
>>> doc = """
```



```

... <div itemscope itemtype="http://schema.org/Product">
...   <span itemprop="name">Kenmore White 17" Microwave</span>
...   
...   <div itemprop="aggregateRating"
...     itemscope itemtype="http://schema.org/AggregateRating">
...     Rated <span itemprop="ratingValue">3.5</span>/5
...     based on <span itemprop="reviewCount">11</span> customer reviews
...   </div>
...
...   <div itemprop="offers" itemscope itemtype="http://schema.org/Offer">
...     <span itemprop="price">$55.00</span>
...     <link itemprop="availability" href="http://schema.org/InStock"
/>In stock
...   </div>
...
...   Product description:
...   <span itemprop="description">0.7 cubic feet countertop microwave.
...   Has six preset cooking categories and convenience features like
...   Add-A-Minute and Child Lock.</span>
...
...   Customer reviews:
...
...   <div itemprop="review" itemscope itemtype="http://schema.org/Review">
...     <span itemprop="name">Not a happy camper</span> -
...     by <span itemprop="author">Ellie</span>,
...     <meta itemprop="datePublished" content="2011-04-01">April 1, 2011
...     <div itemprop="reviewRating" itemscope itemtype="http://schema.org/
Rating">
...       <meta itemprop="worstRating" content = "1">
...       <span itemprop="ratingValue">1</span>/
...       <span itemprop="bestRating">5</span>stars
...     </div>
...     <span itemprop="description">The lamp burned out and now I have to
replace
...     it. </span>
...   </div>
...
...   <div itemprop="review" itemscope itemtype="http://schema.org/Review">

```

```

...     <span itemprop="name">Value purchase</span> -
...     by <span itemprop="author">Lucas</span>,
...     <meta itemprop="datePublished" content="2011-03-25">March 25, 2011
...     <div itemprop="reviewRating" itemscope itemtype="http://schema.org/
Rating">
...         <meta itemprop="worstRating" content = "1"/>
...         <span itemprop="ratingValue">4</span>/
...         <span itemprop="bestRating">5</span>stars
...     </div>
...     <span itemprop="description">Great microwave for the price. It is small and
...     fits in my apartment.</span>
... </div>
...
... </div>
... ""
>>>
>>> for scope in sel.xpath('//div[@itemscope]'):
...     print "current scope:", scope.xpath('@itemtype').extract()
...     props = scope.xpath(''
...         set:difference(./descendant::*/@itemprop,
...             .//*[[@itemscope]/*/@itemprop)''')
...     print "    properties:", props.extract()
...     print
...
current scope: [u'http://schema.org/Product']
    properties: [u'name', u'aggregateRating', u'offers', u'description',
u'review', u'review']

current scope: [u'http://schema.org/AggregateRating']
    properties: [u'ratingValue', u'reviewCount']

current scope: [u'http://schema.org/Offer']
    properties: [u'price', u'availability']

current scope: [u'http://schema.org/Review']
    properties: [u'name', u'author', u'datePublished', u'reviewRating',
u'description']

```



```

current scope: [u'http://schema.org/Rating']
  properties: [u'worstRating', u'ratingValue', u'bestRating']

current scope: [u'http://schema.org/Review']
  properties: [u'name', u'author', u'datePublished', u'reviewRating',
u'description']

current scope: [u'http://schema.org/Rating']
  properties: [u'worstRating', u'ratingValue', u'bestRating']

>>>

```

首先在 `itemscope` 元素上迭代, 对于其中的每一个元素, 我们寻找所有 `itemprops` 元素并排除那些本身在另一个 `itemscope` 中的元素。

### 扩展阅读: XPath的使用提示

#### ➤ 选定条件中的节点文字

当需要将文字内容当作 XPath 的字符串函数的参数时, 应该避免使用 `./text()`, 而应该只用 “.” 代替。这是因为 `./text()` 表达式会生一个文字元素的集合——节点集 (node-set)。当一个节点集合被转换成一个字符串时, 如果将它作为参数传递给一个像 `contains()` 或 `starts-with()` 的字符串函数, 则会产生第一个元素的文本。例如:

```

>>> from scrapy import Selector
>>> sel = Selector(text='<a href="#">Click here to go to the <strong>Next
Page</strong></a>')

```

将一个节点集转换为字符串:

```

>>> sel.xpath('//a/text()').extract() # 查看节点集的内容
[u'Click here to go to the ', u'Next Page']
>>> sel.xpath("string(//a[1]//text())").extract() # 转换为字符串
[u'Click here to go to the ']

```

如果将一个节点对象直接转换为字符串, 则 XPath 会将其下所有子代节点文字的内容一同合并为一个字符串返回:

```
>>> sel.xpath("//a[1]").extract() # 选择第一个节点
[u'<a href="#">Click here to go to the <strong>Next Page</strong></a>']
>>> sel.xpath("string(//a[1])").extract() # 转换为字符串
[u'Click here to go to the Next Page']
```

使用“.”则表示当前节点:

```
>>> sel.xpath("//a[contains(., 'Next Page')]").extract()
[u'<a href="#">Click here to go to the <strong>Next Page</strong></a>']
```

#### ➤ 一定要区分清楚//node[1]和(//node)[1]

- //node[1]——从它们各自的父元素中选择所有位于第1个子节点。
- (//node)[1]——在整个文档中进行选择且只选择第一个匹配的元素。

例如:

```
>>> from scrapy import Selector
>>> sel = Selector(text="""
.....: <ul class="list">
.....:     <li>1</li>
.....:     <li>2</li>
.....:     <li>3</li>
.....: </ul>
.....: <ul class="list">
.....:     <li>4</li>
.....:     <li>5</li>
.....:     <li>6</li>
.....: </ul>""")
>>> xp = lambda x: sel.xpath(x).extract()
```

返回所有具有<li>元素中的第一个<li>元素:

```
>>> xp("//li[1]")
[u'<li>1</li>', u'<li>4</li>']
```

在整个文档中检索所有位于第一位的<li>元素:

```
>>> xp("(//li)[1]")
```



```
[u'<li>1</li>']
```

获取所有在<ul>元素下的第一个<li>元素:

```
>>> xp("//ul/li[1]")
[u'<li>1</li>', u'<li>4</li>']
```

这样就可以在整个文档中获取<ul>中的第一个<li>元素:

```
>>> xp("//ul/li)[1]")
[u'<li>1</li>']
```

#### ➤ 基于类名的选择, 考虑采用CSS选择器而不是XPath

因为一个元素往往会带有一个或多个 CSS 类, 而用 XPath 的方式来以 CSS 类为查询条件选择元素会让代码变得很烦琐, 甚至极为难看:

```
*[contains(concat(' ', normalize-space(@class), ' '), ' someclass ')]
```

如果它们有一个不同的类名称共享字符串 `someclass`, 则使用 `@class='someclass'` 可能会丢失具有其他类的元素, 如果只使用 `contains(@class, 'someclass')` 来弥补, 则可能会得到更多的元素。

事实证明, Scrapy 选择器可以使用链式选择器, 所以大多数情况下可以通过 CSS 来选择类, 然后在需要时切换到 XPath:

```
>>> from scrapy import Selector
>>> sel = Selector(text='<div class="hero shout"><time datetime="2014-07-23
19:00">Special date</time></div>')
>>> sel.css('.shout').xpath('./time/@datetime').extract()
[u'2014-07-23 19:00']
```

这比单纯地使用 XPath 技巧更清晰明了, 同时可以结合两者的优点。

## 4.4.4 非结构化数据的提取

计算机系统中的数据分为结构化数据和非结构化数据。结构化数据具有标准的数据类型, 结构完整、格式规范, 像 HTML、XML、XSL、JSON 及 SQL 数据库都属于结构化数据。非结构化数据的格式非常多样, 标准也是多样性的, 而且在技术上非结构化信息比结构化信息更难

标准化和理解。非结构化数据是指数据结构不规则或不完整，没有预定义的数据模型，不方便用数据库二维逻辑表来表现的数据。

### 应用场景

据 IDC 的一项调查报告中指出：企业中 80% 的数据都是非结构化数据，这些数据每年都按指数增长 60%。据报道指出：平均只有 1%~5% 的数据是结构化的数据。如今，这种迅猛增长的从不使用的数据在企业中消耗着复杂且昂贵的一级存储的存储容量。如何更好地保留那些在全球范围内具有潜在价值的不同类型的文件，而不是因为处理它们却干扰日常的工作？云存储是越来越多的 IT 公司正在使用的存储技术。

这些非结构化数据常存在于以下文件格式中：

- PDF、Rtf、Word、Excel 等具有复杂格式的办公文件；
- 以逗号分隔的文本（CSV）或纯文本（txt）文件；
- 图片、视频及其他多媒体文件格式；
- 电子邮件。

### 办公文件

当我们去爬取一些现有的知识库系统（如百度文库）时，就有可能需要从 PDF、Word、Rtf、Excel 和 PowerPoint 等格式的文档中提取出可以描述文档的文字，这些描述性的信息包括文档标题、作者、主要内容等。而更为复杂的情况可能是要从这些文档所包含的表格中取出数据并进行查询汇总之用。

### 纯文本文件

现在还有不少企业仍然会采用纯文本，例如，.txt、.csv 文件，以逗号分隔的方式保存一些表格数据并以此作为企业间交换数据的方式。这种做法由来已久，因为文本文件最大的优势就是兼容性强，任何操作系统无须额外的工具就能查看纯文本的内容，而且格式也容易理解与输出。但逗号分隔的文本文件也具有以下通病：

- 可能带有特定的编码，内容中容易出现乱码；
- 没有特定的数据类型（从 Python 中读出的都是字符串），当读取时需要进行类型转换与校验；
- 可能存在大量的空值；
- 如果通过列名标识字段，也容易产生乱码的问题；
- 通过列索引来标识字段，当文本取自不同来源时，可能存在字段位置不统一的情况；
- 全角与半角的数字符号容易产生倒置数据类型的错误，例如，“”和“”。



## 图片及多媒体文件

还有一些情况是爬取的目标数据或文字存在于图片上, 要求蜘蛛具有图片识别能力, 能准确地从图片的某个区域中将图形读出并转化为相应的内容。典型的用法就是某些网站上要求用户输入的图形识别码。

### 4.4.4.1 正则表达式

上一节简单地提了一下如何在选择器中应用正则表达式, 但并没有详细展开。在结构化的文档中使用正则表达式的机会并不多, 因为它本身非常难以理解, 而且结构化文档可以有很多取代正则表达式的简单方法。但是在非结构化文件中, 尤其是纯文字内容的文件, 要从中定位某个具有一定标志性的内容时, 正则表达式的地位就显得尤为重要了。

接下来介绍正则表达式的用法, 并结合 Python 应用到程序代码中。

正则表达式 (regular expression) 描述了一种字符串匹配的模式 (pattern), 可以用来检查一个串是否含有某种子串, 将匹配的子串替换, 或者从某个串中取出符合某个条件的子串等。

构造正则表达式的方法和创建数学表达式的方法一样。也就是用多种元字符与运算符将小的表达式结合在一起来创建更大的表达式。正则表达式的组件可以是单个的字符、字符集合、字符范围、字符间的选择, 或者所有这些组件的任意组合。

正则表达式是由普通字符 (例如, 字符 a 到 z) 和特殊字符 (称为“元字符”) 组成的文字模式。模式描述在搜索文本时要匹配一个或多个字符串。正则表达式作为一个模板, 将某个字符模式与所搜索的字符串进行匹配。

#### 普通字符

普通字符包括没有显式指定为元字符的所有可打印和不可打印字符, 包括所有大写和小写字母、所有数字、所有标点符号和一些其他符号。

#### 非打印字符

非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列。

字 符	描 述
\cx	匹配由 x 指明的控制字符。例如, \cM 匹配一个 Control-M 或回车符。x 的值必须为 A~Z 或 a~z 之一。否则, 将 c 视为一个原义的 'c' 字符
\f	匹配一个换页符, 等价于 \x0c 和 \cL
\n	匹配一个换行符, 等价于 \x0a 和 \cJ

续表

字 符	描 述
\r	匹配一个回车符，等价于\x0d和\cM
\s	匹配任何空白字符，包括空格、制表符、换页符等，等价于[ \f\n\r\t\v]
\S	匹配任何非空白字符，等价于[^ \f\n\r\t\v]
\t	匹配一个制表符，等价于\x09和\cI
\v	匹配一个垂直制表符，等价于\x0b和\cK

### 特殊字符

所谓特殊字符，就是指一些有特殊含义的字符，如上面 `runoo*b` 中的“\*”，简单地讲就是表示任何字符串的意思。如果要查找字符串中的“\*”符号，则需要对“\*”进行转义，即在其前面加一个“\”：`runo\*ob` 匹配 `runo*ob`。

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须先使字符“转义”，即将反斜杠字符“\”放在它们前面。下表列出了正则表达式中的特殊字符。

特 别 字 符	描 述
\$	匹配输入字符串的结尾位置。如果设置了 <code>RegExp</code> 对象的 <code>Multiline</code> 属性，则\$也匹配'\n'或'\r'。要匹配“\$”字符本身，则使用“\\$”
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，则使用“\(”和“\)”
*	匹配前面的子表达式零次或多次。要匹配“*”字符，则使用“\*”
+	匹配前面的子表达式一次或多次。要匹配“+”字符，则使用“\+”
.	匹配除换行符\n外的任何单字符。要匹配“.”，则使用“\.”
[	标记一个中括号表达式的开始。要匹配“[”，则使用“\[”
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配“?”字符，则使用“\?”
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如，'\n'匹配字符'n'。'\n'匹配换行符。序列'\\'匹配“\”，而'\('匹配“(”
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配“^”字符本身，则使用“\^”
{	标记限定符表达式的开始。要匹配“{”，则使用“\{”
	指明两项之间的一个选择。要匹配“ ”，则使用“\ ”。

例如，由于章节编号在大的输入文档中很可能超过 9，所以需要一种方式来处理两位或三



位章节编号——利用限定符给可以实现。下面的正则表达式匹配编号为任何位数的章节标题:

```
/Chapter [1-9][0-9]*/
```

注意, 限定符出现在范围表达式之后。因此, 它应用于整个范围表达式, 在本例中, 只指定从 0 到 9 的数字 (包括 0 和 9)。这里不使用 “+” 限定符, 因为在第二个位置或后面的位置不一定需要一个数字。也不使用 “?” 字符, 因为使用 “?” 会将章节编号限制为只有两位数。需要至少匹配 Chapter 和空格字符后面的一个数字。如果知道章节编号被限制为只有 99 章, 可以使用下面的表达式来至少指定一位但至多两位数字。

```
/Chapter [0-9]{1,2}/
```

上面的表达式的缺点是大于 99 的章节编号仍只匹配开头两位数字。另一个缺点是 Chapter 0 也将匹配。只匹配两位数字的更好的表达式如下:

```
/Chapter [1-9][0-9]?/
```

或

```
/Chapter [1-9][0-9]{0,1}/
```

“\*”、“+” 限定符都是贪婪的, 因为它们会尽可能多地匹配文字, 在它们的后面加上一个 “?” 就可以实现非贪婪或最小匹配。

例如, 搜索 HTML 文档以查找 H1 标记内的章节标题。该文本在文档中如下:

```
<H1>Chapter 1 - 介绍正则表达式</H1>
```

贪婪: 下面的表达式匹配从开始小于符号 (<) 到关闭 H1 标记的大于符号 (>) 之间的所有内容。

```
/<.*>/
```

非贪婪: 如果只需要匹配开始和结束的 H1 标签, 则下面的非贪婪表达式只匹配 <H1>。

```
/<.*?>/
```

如果只想匹配开始的 H1 标签, 则表达式是:

```
/<\w+?>/
```

在“\*”、“+”或“?”限定符之后放置“?”,该表达式从“贪心”表达式转换为“非贪心”表达式或者最小匹配。

## 定位符

定位符能够将正则表达式固定到行首或行尾。它们还能够创建出现在一个单词内、一个单词的开头或者一个单词的结尾的正则表达式。

定位符用来描述字符串或单词的边界,“^”和“\$”分别指字符串的开始与结束,“\b”描述单词的前或后边界,“\B”表示非单词边界。

正则表达式的定位符如下表所示。

字 符	描 述
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性,则“^”还会与“\n”或“\r”之后的位置匹配
\$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性,则“\$”还会与“\n”或“\r”之前的位置匹配
\b	匹配一个字边界,即字与空格间的位置
\B	非字边界匹配

**注意:** 不能将限定符与定位符一起使用。由于在紧靠换行或者字边界的前面或后面不能有一个以上的位置,因此不允许诸如“^、\*”之类的表达式。

- 若要匹配一行文本开始处的文本,则在正则表达式的开始使用“^”字符。不要将“^”这种用法与中括号表达式中的用法混淆。
- 若要匹配一行文本的结束处的文本,则在正则表达式的结束处使用“\$”字符。
- 若要在搜索章节标题时使用定位点,则下面的正则表达式匹配一个章节标题,该标题只包含两个尾随数字,并且出现在行首:

```
/^Chapter [1-9][0-9]{0,1}/
```

真正的章节标题不仅出现行的开始处,而且它还是该行中仅有的文本。它既出现在行首,又出现在同一行的结尾。下面的表达式能确保指定的匹配只匹配章节而不匹配交叉引用。通过创建只匹配一行文本的开始和结尾的正则表达式就可做到这一点。



```
/^Chapter [1-9][0-9]{0,1}$/
```

匹配字边界稍有不同,但给正则表达式增加了很重要的能力。字边界是单词和空格之间的位置。非字边界是任何其他位置。下面的表达式匹配单词 `Chapter` 的开头三个字符,因为这三个字符出现字边界后面:

```
/\bCha/
```

`\b` 字符的位置是非常重要的。如果它位于要匹配的字符串的开始,则它在单词的开始处查找匹配项。如果它位于字符串的结尾,则它在单词的结尾处查找匹配项。例如,下面的表达式匹配单词 `Chapter` 中的字符串 `ter`,因为它出现在字边界的前面:

```
/ter\b/
```

下面的表达式匹配 `Chapter` 中的字符串 `apt`,但不匹配 `aptitude` 中的字符串 `apt`:

```
/\Bapt/
```

字符串 `apt` 出现在单词 `Chapter` 中的非字边界处,但出现在单词 `aptitude` 中的字边界处。`\B` 非字边界运算符的位置并不重要,因为匹配不关心究竟是单词的开头还是结尾。

## 选择

用圆括号将所有选择项括起来,相邻的选择项之间用“`|`”分隔。但用圆括号会有一个副作用,会使相关的匹配被缓存,此时可用“`?:`”放在第一个选项前来消除这种副作用。其中“`?:`”是非捕获元之一,还有两个非捕获元是“`?=`”和“`?!`”。这两个非捕获元还有更多的含义,前者为正向预查,在任何开始匹配圆括号内的正则表达式模式的位置来匹配搜索字符串;后者为负向预查,在任何开始不匹配该正则表达式模式的位置来匹配搜索字符串。

## 反向引用

为一个正则表达式模式或部分模式两边添加圆括号将导致相关匹配存储到一个临时缓冲区中,所捕获的每个子匹配都按照在正则表达式模式中从左到右出现的顺序存储。缓冲区编号从 1 开始,最多可存储 99 个捕获的子表达式。每个缓冲区都可以使用 `\n` 访问,其中 `n` 标识特定缓冲区的一位或两位十进制数。

可以使用非捕获元字符“`?:`”、“`?=`”或“`?!`”来重写捕获,忽略对相关匹配的保存。

反向引用可以查找文本中两个相同的相邻单词的匹配项。以下的句子为例:

```
Is is the cost of of gasoline going up up?
```

上面的句子很显然有多个重复的单词。如果设计一种方法能定位该句子，而不必查找重复出现的每个单词，那该有多好啊。下面的正则表达式使用单个子表达式来实现这一点：

```
>>> import re
>>> sentence = "Is is the cost of of gasoline going up up"
>>> partt= r'\\b([a-z]+) \\b/'
>>> print re.match(partt,sentence)
'Is is,of of,up up'
```

捕获的表达式正如[\[a-z\]+](#)指定的，包括一个或多个字母。正则表达式的第二部分是对以前捕获的子匹配项的引用，即单词的第二个匹配项正好由括号表达式匹配。`\\b`指定第一个子匹配项。字边界元字符确保只检测整个单词。否则，诸如“is issued”或“this is”之类的词组将不能正确地由此表达式识别。正则表达式后面的全局标记 `g` 指定将该表达式应用到输入字符串中能够查找到尽可能多的匹配。表达式的结尾处的 `i` 标记指定不区分大小写。多行标记指定换行符的两边可能出现潜在的匹配。

反向引用还可以将通用资源指示符（URI）分解为其组件。假定想将下面的 URI 分解为协议（FTP、HTTP 等）、域地址和页/路径：

```
http://www.example.com:80/html/html-tutorial.html
```

下面的正则表达式提供了该功能：

```
>>> import re
>>> url = "http://www.example.com:80/html/html-tutorial.html"
>>> patt = r'/(\\w+):\\/\\/([^/:]+)(:\\d*)?([^# ]*)/'
>>> results = re.match(patt,url)
>>> for result in results:
>>>     print result
'http://www.example.com:80/html/html-tutorial.html'
'http'
'www.example.com'
':80'
'/html/html-tutorial.html'
```

- 第一个括号子表达式捕获 Web 地址的协议部分。该子表达式匹配在冒号和两个正斜杠前面的任何单词。



- 第二个括号子表达式捕获地址的域地址部分。子表达式匹配 “/” 和 “:” 之外的一个或多个字符。
- 第三个括号子表达式捕获端口号（如果指定了）。该子表达式匹配冒号后面的零个或多个数字。只能重复一次该子表达式。
- 第四个括号子表达式捕获 Web 地址指定的路径和 “/” 或页信息。该子表达式能匹配不包括 “#” 或空格字符的任何字符序列。

将正则表达式应用到上面的 URI，各子匹配项包含下面的内容：

- 第一个括号子表达式包含 “http”；
- 第二个括号子表达式包含 “www.example.com”；
- 第三个括号子表达式包含 “:80”；
- 第四个括号子表达式包含 “/html/html-tutorial.html”。

Python的正规表达式用法

Python 自 1.5 版本起增加了 `re` 模块，它提供 Perl 风格的正则表达式模式。`re` 模块使 Python 语言拥有全部的正则表达式功能。`compile` 函数根据一个模式字符串和可选的标志参数生成一个正则表达式对象。该对象拥有一系列方法用于正则表达式匹配和替换。`re` 模块也提供了与这些方法功能完全一致的函数，这些函数使用一个模式字符串作为它们的第一个参数。

➤ `re.match`函数

`re.match` 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功，则 `match()` 返回 `None`。

函数语法：

```
re.match(pattern, string, flags=0)
```

函数参数说明如下表所示。

参 数	描 述
pattern	匹配的正则表达式
string	要匹配的字符串
flags	标志位，用于控制正则表达式的匹配方式，如是否区分大小写、多行匹配等

匹配成功则 `re.match` 方法返回一个匹配的对象，否则返回 `None`。

示例:

```
>>> import re
>>> print(re.match('www', 'www.example.com').span())    # 在起始位置匹配
>>> print(re.match('com', 'www.example.com'))           # 不在起始位置匹配
(0, 3)
None
```

我们可以使用 `group(num)` 或 `groups()` 匹配对象函数来获取匹配表达式, 如下表所示。

匹配对象方法	描 述
<code>group(num=0)</code>	匹配的整个表达式的字符串, <code>group()</code> 可以一次输入多个组号, 在这种情况下它将返回一个包含那些组所对应值的元组
<code>groups()</code>	返回一个包含所有小组字符串的元组, 从 1 到所含的小组号

示例:

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
    print "matchObj.group() : ", matchObj.group()
    print "matchObj.group(1) : ", matchObj.group(1)
    print "matchObj.group(2) : ", matchObj.group(2)
else:
    print "No match!!"
```

结果输出如下:

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter
```



➤ re.search方法

re.search 扫描整个字符串并返回第一个成功的匹配。

函数语法:

```
re.search(pattern, string, flags=0)
```

函数参数说明如下表所示。

参 数	描 述
pattern	匹配的正则表达式
string	要匹配的字符串
flags	标志位，用于控制正则表达式的匹配方式，如是否区分大小写、多行匹配等

匹配成功则 re.search 方法返回一个匹配的对象，否则返回 None。

示例:

```
>>> import re
>>> print(re.search('www', 'www.example.com').span()) # 在起始位置匹配
>>> print(re.search('com', 'www.example.com').span()) # 不在起始位置匹配
(0,3)
(12,15)
```

我们可以使用 group(num) 或 groups() 匹配对象函数来获取匹配表达式，如下表所示。

匹配对象方法	描 述
group(num=0)	匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组
groups()	返回一个包含所有小组字符串的元组，从 1 到所含的小组号

示例:

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";
```

```
searchObj = re.search( r'(.*) are (.*) .*', line, re.M|re.I)

if searchObj:
    print "searchObj.group() : ", searchObj.group()
    print "searchObj.group(1) : ", searchObj.group(1)
    print "searchObj.group(2) : ", searchObj.group(2)
else:
    print "Nothing found!!"
```

以上实例执行结果如下：

```
searchObj.group() : Cats are smarter than dogs
searchObj.group(1) : Cats
searchObj.group(2) : smarter
```

#### ➤ re.match与re.search的区别

re.match 只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回 None。而 re.search 匹配整个字符串，直到找到一个匹配。

示例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
    print "match --> matchObj.group() : ", matchObj.group()
else:
    print "No match!!"

matchObj = re.search( r'dogs', line, re.M|re.I)
if matchObj:
    print "search --> matchObj.group() : ", matchObj.group()
else:
    print "No match!!"
```



以上实例运行结果如下:

```
No match!!
search --> matchObj.group() : dogs
```

➤ 检索和替换

Python 的 re 模块提供了 re.sub, 用于替换字符串中的匹配项。

语法:

```
re.sub(pattern, repl, string, count=0, flags=0)
```

参数说明如下表所示。

参 数	说 明
pattern	正则中的模式字符串
repl	替换的字符串, 也可为一个函数
string	要被查找替换的原始字符串
count	模式匹配后替换的最大次数, 默认为 0, 表示替换所有的匹配

➤ 实例

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import re

phone = "2004-959-559 # 这是一个国外电话号码"

# 删除字符串中的 Python 注释
num = re.sub(r'#.*$', "", phone)
print "电话号码是: ", num

# 删除非数字 (-) 的字符串
num = re.sub(r'\D', "", phone)
print "电话号码是 : ", num
```

以上实例执行结果如下:

电话号码是： 2004-959-559

电话号码是： 2004959559

### ➤ 替换

以下实例将字符串中匹配的数字乘以 2：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import re

# 将匹配的数字乘以 2
def double(matched):
    value = int(matched.group('value'))
    return str(value * 2)

s = 'A23G4HFD567'
print(re.sub('(P<value>\d+)', double, s))
```

执行输出结果为：

A46G8HFD1134

### ➤ 正则表达式修饰符——可选标志

正则表达式可以包含一些可选标志修饰符来控制匹配的模式。修饰符被指定为一个可选的标志。多个标志可以通过按位 OR (|) 来指定，如 `re.I | re.M` 被设置成 `I` 和 `M` 标志。修饰符如下表所示。

修 饰 符	描 述
<code>re.I</code>	使匹配对大小写不敏感
<code>re.L</code>	做本地化识别 (locale-aware) 匹配
<code>re.M</code>	多行匹配，影响 <code>^</code> 和 <code>\$</code>
<code>re.S</code>	使 “.” 匹配包括换行在内的所有字符
<code>re.U</code>	根据 Unicode 字符集解析字符。这个标志影响 <code>\w</code> 、 <code>\W</code> 、 <code>\b</code> 、 <code>\B</code>
<code>re.X</code>	该标志通过更灵活的格式使得正则表达式可以写得更易于理解



#### 4.4.4.2 读取CSV文件

在某些应用场景下, CSVFeedSpider 可能无法满足我们爬取 CSV 文件的要求, 此时就需要手工读取 CSV 文件中的数据。Python 提供了内置 CSV 库来处理 CSV 文件, 先看一下 Python 是如何直接处理 CSV 文件的, 假设在当前的工作目录下有一份名为 “my-books.csv” 的文件:

```
书名,作者,出版年份
《Vue2 实践揭秘》,梁睿坤,2017
《虫术》,梁睿坤,2017
```

CSV 的第一行通常都是列名 (Column Name/Field Name), 当然这只是 “一般性” 做法, 因为没有列名的 CSV 文件也是可以的。在 Python 中可以用以下方法将其读取并打印到屏幕上:

```
import csv
with open('my-books.csv') as book_file:
    csv_reader = csv.reader(book_file)
    for row in csv_reader:
        print row
```

其输出结果如下:

```
['\\xe4\\xb9\\xa6\\xe5\\x90\\x8d', '\\xe4\\xbd\\x9c\\xe8\\x80\\x85', '\\xe5\\x87\\xba\\x
e7\\x89\\x88\\xe5\\xb9\\xb4\\xe4\\xbb\\xbd']
['\\xe3\\x80\\x8aVue2\\xe5\\xae\\x9e\\xe8\\xb7\\xb5\\xe6\\x8f\\xad\\xe7\\xa7\\x98\\xe3\\x
80\\x8b', '\\xe6\\xa2\\x81\\xe7\\x9d\\xbf\\xe5\\x9d\\xa4', '2017']
['\\xe3\\x80\\x8a\\xe8\\x99\\xab\\xe6\\x9c\\xaf\\xe3\\x80\\x8b', '\\xe6\\xa2\\x81\\xe7\\
x9d\\xbf\\xe5\\x9d\\xa4', '2017']
```

csv 库是 Python 内置的, 它提供了一些针对 CSV 文件处理的读取器。例如, 上述代码中的 reader 工厂方法, 它就是构造出的一个可枚举的读取器, 用于迭代 CSV 文件中的行。然而, 为何 Python 会将中文变成一堆我们看不懂的符号呢? 这是由于中文是一种非 ASCII 字符, 每个中文需要两个英文字母来表示, 因此只能通过 Base64 或者 UTF-8 的方式才能对其解码显示:

```
import csv
from io import StringIO
with open('my-books.csv').read().decode('utf8') as book_data:
    csv_reader = csv.reader(StringIO(book_data))
```

```
for row in csv_reader:
    print row
```

输出结果:

```
[u'书名',u'作者',u'出版年份']
[u'《Vue2 实践揭秘》',u'梁睿坤','2017']
[u'《虫术》',u'梁睿坤','2017']
```

由于我们要对文字编码进行解码操作,因此需要先将文字内容读出后才能调用 `decode` 方法。当完成解码后,得到的是一个 `Unicode` 字符串。所以如果能让 `Python` 操作字符串时仍然能像操作文件一样,就需要 `io.StringIO` 类对字符串进行一次包装, `StringIO` 对象能将字符串对象的操作接口包装成与文件操作接口相一致,这样就可以将其直接作为输入参数放进 `reader` 方法中。这是 `Python` 中将字符串作为文件操作的一个常用的技巧。

上述做法只能得到行值的数组,在使用时不得不采用索引来引用具体的字段值:

```
for row in csv_reader:
    print u'书名:%s' % row[1]
```

而且这样做字段名与值是没有直接关联关系的, `Python` 不会让我们用这么笨的方法来处理一件事,它提供了一个称为 `DictReader` 的类,可以将字段名与值放在字典中对应起来。用 `DictReader` 读取 `my-books.csv` 文件,代码如下所示。

```
import csv
from io import StringIO
with open('my-books.csv').read().decode('utf8') as book_data:
    dict_reader = csv.DictReader(StringIO(book_data))
    print dict_reader.fieldnames # 打印字段名
    for row in dict_reader:
        print row
```

其输出结果如下:

```
[u'书名',u'作者',u'出版年份']
```

## 在爬虫中处理CSV文件

以上是本地处理文件的操作方式,当 `CSV` 文件是一个网络资源时,则需要先行下载:



```

from urllib.request import urlopen
from io import StringIO
import csv

data = urlopen("http://www.example.com/files/my-books.csv").read().decode(
    'utf8')
dataFile = StringIO(data)
dictReader = csv.DictReader(dataFile)

for row in dictReader:
    print(row)

```

当然,上述代码只是为了能让我们能得到一个可以运行的示例而已,在真正的爬虫系统中,我们一定不会这么做,因为下载的动作 **Scrapy** 已让下载器完成,CSV 的返回值会被保存在 `Response.body` 属性中,那么我们就可以用以下方式来获取 CSV 的读取器实例,代码如下所示。

```

import csv
class MyCSVSpider(Spider):
    # ... 省略

    def parse(response):
        csv_data = StringIO(response.body.decode('utf8'))
        dictReader = csv.DictReader(csv_data)
        # ... 省略

```

#### 4.4.4.3 读取Excel

Python 直接读取 Excel 格式的文件主要有 `xlrd`、`xlutils`、`openpyxl` 几种主流的库可用,其中 `xlrd` 提供的对象模型是比较容易上手的。以下是 `xlrd` 的一个简单的使用示例:

```

import xlrd
workbook = xlrd.open_workbook(u'每日数据及趋势.xls')
sheet_names= workbook.sheet_names()

for sheet_name in sheet_names:
    sheet2 = workbook.sheet_by_name(sheet_name)
    print sheet_name

```

```
rows = sheet2.row_values(3) # 获取第四行内容
cols = sheet2.col_values(1) # 获取第二列内容

print rows
print cols
```

### 转换为CSV处理

如果采用 `xlrd` 这类 Excel 兼容库, 则只能通过索引获得具体值, 这会使整个处理过程变得非常麻烦。而且这些库都必须将 Excel 文件中的结构对象实例化。与其采用这种复杂而低效的处理方式, 不如将 Excel 直接转换成 CSV 文件, 如前文一样采用 `DictReader` 处理转化后的 Excel 会更简单。

以下示例将爬取到的响应正文中的内容读取至文件, 然后用 `Xlsx2csv` 对文件进行转换, 再将其读取为 CSV 字典对象, 具体代码示例如下:

可以通过 `$ pip install xlsx2csv` 下载安装 `Xlsx2csv` 工具包。

```
import csv
import uuid
import os
from scrapy import Spider
from xlsx2csv import Xlsx2csv

class MyExcelSpider(Spider):

    def parse(self, response):
        dataFile = StringIO(response.body)
        tmp_file_name = os.path.join('/tmp', '%s.csv' % uuid.uuid4())

        with open(tmp_file_name, 'wb') as tmp_csv_file: # 将结果保存到临时文件内
            csv_file = Xlsx2csv(dataFile).convert(tmp_csv_file, sheetid=1)
            reader = csv.DictReader(csv_file) # 获取 CSV 的字典读取器

            # ... 加入对 CSV 文件内容的处理 (省略)

        os.remove(tmp_file_name) # 移除临时文件
```



#### 4.4.4.4 读取PDF文件

PDF 是比较棘手的通用网络文件, 首先 PDF 的文件尺寸比较大, 通常每个文件的大小量级都在 MB 以上, 其次 PDF 的文件内容都是图片, 对其内容进行读取实际上是一个图片识别的过程。就以上两点来说, PDF 是不适合在蜘蛛内进行直接处理的, 因为这样做会让爬虫系统成为一个吞噬内存与 CPU 的“怪兽”, 一旦爬取的文件数量过大, 可能会导致系统因资源消耗过量而宕机。

因此, 对于 PDF 文件类型的处理应该在爬虫系统完成数据爬取后启用独立的 Python 进程对其进行专门的分析与读取。

以下是 Python 社区中比较出名和常用的 PDF 工具包:

- PDFMiner (<https://github.com/euske/pdfminer>)——一个从 PDF 文档中提取信息的工具。
- PyPDF2 (<https://github.com/mstamy2/PyPDF2>)——一个能够分割、合并和转换 PDF 页面的库。
- pdftables (<https://pypi.python.org/pypi/pdftables>)——直接从 PDF 文件中提取表格。

此处就以 PDFMiner 工具包为例, 将一个 PDF 文件中的所有文本内容读出并转化为文字:

```
from urllib.request import urlopen
from pdfminer.pdfinterp import PDFResourceManager, process_pdf
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from io import StringIO
from io import open

def readPDF(pdfFile):
    rsrcmgr = PDFResourceManager()
    retstr = StringIO()
    laparams = LAParams()
    device = TextConverter(rsrcmgr, retstr, laparams=laparams)
    process_pdf(rsrcmgr, device, pdfFile)
    device.close()

    content = retstr.getvalue()
    retstr.close()

    return content
```

```
pdfFile = urlopen("http://domain.com/somefile.pdf");  
outputString = readPDF(pdfFile)  
  
print(outputString)  
pdfFile.close()
```

#### 4.4.4.5 读取Microsoft Word和.docx文件

如果论最通用的非结构化文档格式，则 Word 文档在数量上可以算是世界第一了。因此在对非结构化文档的采集场景中，Word 文档格式是最常见的一种，我们有必要了解一些在 Python 中读取 Word 文档的手段。

python-docx (<https://python-docx.readthedocs.io/en/latest/index.html>) 是一款比较实用的操作 Word 文档的 Python 工具包，它体积很少，功能也相对齐全。在运行时，内存与 CPU 的消耗量也相对较小。

##### 安装

可以按以下指令安装 python-docx：

```
$ pip install python-docx
```

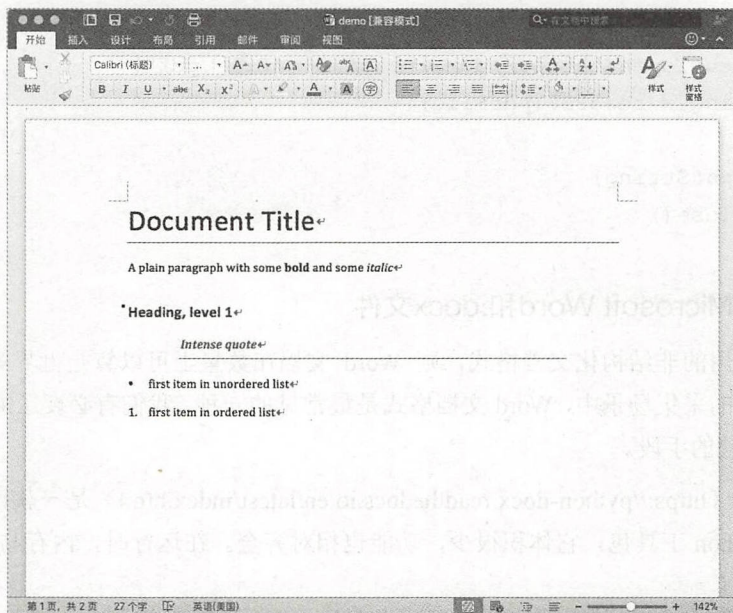
安装完成后，在代码中将 docx 包导入即可使用：

```
import docx
```

##### 读取Word中的文本

先建立一个简单的范例文本作为测试之用，具体内容如下图所示。





首先要了解 Word 文档的最基本结构：段落。这是构成 Word 文档的基本要素，而组成段落的更小元素是运行对象（run）。在 `python-docx` 中读取段落或者运行对象的方法都是一样的。

例如，计算 Word 文档中共有多少段落，可以用以下方法：

```
doc = docx.Document('demo.docx')
print len(doc.paragraphs)
```

我们必须从 `Document` 对象的 `paragraphs` 段落集合对象中获取具体的段落。例如，要知道每个段落是由多少个运行对象组成的，可以使用以下代码：

```
doc = docx.Document('demo.docx')
for p in doc.paragraphs:
    print len(p.runs)
```

读取段落或者运行对象中的具体文字内容也是相当简单的，它们都具有 `text` 的成员属性，直接读取该属性即可获取其内部的文字。以下是将前文中整个 Word 文档内容一次性读取的代码：

```
# coding:utf-8
import docx
```

```
doc = docx.Document('demo.docx')
print "文档中共发现%s 个段落" % len(doc.paragraphs)

if len(doc.paragraphs):
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
    print '\n'.join(fullText)
```

输出:

```
文档中共发现 7 个段落
Document Title
A plain paragraph with some bold and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list
```

虽说 `python-docx` 比较受欢迎,但由于 Word 文档的历史太悠久了,内容也非常复杂,`python-docx` 只能读取一些比较简单常用的格式文档。如果遇到结构非常复杂的 Word 文档可能就不太适用,对于复杂的使用场景,可以将 Word 文件以 zip 文件格式打开,然后直接读取它的 XML,此种做法需要对 Word 文件的 Schema 结构非常熟悉,也不失为一种可以用于攻坚的应急手段。

`python-docx` 的更多内容在此就不细说了,详情可以参考官方 API。

### 4.4.5 黑夜中的眼睛

黑夜给我了一双黑色的眼睛,我却用它们寻找光明。

网络中的请求与响应是机器之间的交互,它们完全是自动运行的。我们产生的只是一个触发行而已,如输入某个网址或者单击某个链接,那么请求与响应就“自动”产生了。对于我们来说,这简直就像是置身于毫无光明的黑暗之中,我们对背后的动作情况可以说是一无所知。

越是了解网络背后发生的事情,就越有利于爬虫系统开发,这就需要我们找到一些有力的工具为开发助力。



## curl

curl 是一种命令行工具，作用是发出网络请求，然后得到并提取数据，显示在“标准输出”（stdout）上面。curl 可谓是爬虫开发中必不可少的轻量级工具了，我们可以用它在命令行中模拟出很多请求效果，在着手写代码前观察响应的结果等。

curl 支持多种协议，下面举例简单介绍它的使用。

### ➤ 查看网页源码

直接在 curl 命令后加上网址，就可以看到网页源码。我们以网址 [www.sina.com](http://www.sina.com) 为例（选择该网址，主要因为它的网页代码较短）：

```
$ curl www.sina.com

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://www.sina.com.cn/">here</a>.</p>
</body></html>
```

如果要把这个网页保存下来，则可以使用 -o 参数，相当于使用 wget 命令。

```
$ curl -o [文件名] www.sina.com
```

### ➤ 自动跳转

有的网址是自动跳转（重定向）的。使用 -L 参数，curl 就会跳转到新的网址。

```
$ curl -L www.sina.com
```

键入上面的命令，结果就自动跳转为 [www.sina.com.cn](http://www.sina.com.cn)。

### ➤ 显示头信息

-i 参数可以显示 HTTP response 的头信息，连同网页代码一起：

```
$ curl -i www.sina.com
```

```
HTTP/1.0 301 Moved Permanently
```

```

Date: Sat, 03 Sep 2011 23:44:10 GMT
Server: Apache/2.0.54 (Unix)
Location: http://www.sina.com.cn/
Cache-Control: max-age=3600
Expires: Sun, 04 Sep 2017 00:44:10 GMT
Vary: Accept-Encoding
Content-Length: 231
Content-Type: text/html; charset=iso-8859-1
Age: 3239
X-Cache: HIT from sh201-9.sina.com.cn
Connection: close

```

```

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://www.sina.com.cn/">here</a>.</p>
</body></html>

```

-I 参数则是只显示 HTTP response 的头信息:

```

$ curl -I -L http://www.jd.com/2017
HTTP/1.1 302 Moved Temporarily
Server: JDWS/2.0
Date: Tue, 02 Jan 2018 05:05:37 GMT
Content-Type: text/html
Content-Length: 157
Connection: keep-alive
Location: https://www.jd.com/2017
Strict-Transport-Security: max-age=360

HTTP/1.1 200 OK
Server: JDWS/2.0
Date: Tue, 02 Jan 2018 05:05:37 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 114586
Connection: keep-alive

```



```
Vary: Accept-Encoding
Vary: Accept-Encoding
Expires: Tue, 02 Jan 2018 05:06:05 GMT
Cache-Control: max-age=30
ser: 4.126
Via: BJ-M-YZ-NX-80 (HIT), http/1.1 GZ-CT-1-JCS-39 ( [cMsSf ])
Age: 0
Strict-Transport-Security: max-age=360
```

在分析响应头时作用特别大, 而且相当好用。

### ➤ 显示通信过程

-v 参数可以显示一次 HTTP 通信的整个过程, 包括端口连接和 HTTP request 头信息:

```
$ curl -v http://www.baidu.com
* Rebuilt URL to: http://www.baidu.com/
* Trying 14.215.177.39...
* Connected to www.baidu.com (14.215.177.39) port 80 (#0)
> GET / HTTP/1.1
> Host: www.baidu.com
> User-Agent: curl/7.47.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: bfe/1.0.8.18
< Date: Tue, 02 Jan 2018 05:07:38 GMT
< Content-Type: text/html
< Content-Length: 2381
< Last-Modified: Mon, 23 Jan 2017 13:27:56 GMT
< Connection: Keep-Alive
< ETag: "588604dc-94d"
< Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform
< Pragma: no-cache
< Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/
< Accept-Ranges: bytes
....
</div> </div> </div> </body> </html>
* Connection #0 to host www.baidu.com left intact
```

如果觉得上面的信息还不够，那么下面的命令可以将整个通信过程的详细信息保存到 output.txt 文件中。

```
$ curl --trace output.txt www.baidu.com
```

或者

```
$ curl --trace-ascii output.txt www.baidu.com
```

运行后，打开 output.txt 文件查看。

### ➤ 发送表单信息

发送表单信息有 GET 和 POST 两种方法。GET 方法相对简单，只要把数据附在网址后面就行。

```
$ curl example.com/form.cgi?data=xxx
```

POST 方法必须把数据和网址分开，curl 要用到 --data 参数。

```
$ curl -X POST --data "data=xxx" example.com/form.cgi
```

如果数据没有经过表单编码，则可以让 curl 进行编码，参数是 --data-urlencode。

```
$ curl -X POST --data-urlencode "date=April 1" example.com/form.cgi
```

### ➤ HTTP 方法

curl 默认的 HTTP 方法是 GET，使用 -x 参数可以支持其他方法。

```
$ curl -X POST www.example.com
```

```
$ curl -X DELETE www.example.com
```

### ➤ 文件上传

假定文件上传的表单如下：

```
<form method="POST" enctype='multipart/form-data' action="upload.cgi">  
<input type=file name=upload>
```



```
<input type=submit name=press value="OK">
</form>
```

可以用 curl 上传文件:

```
$ curl --form upload=@localfilename --form press=OK [URL]
```

### ➤ Referer 字段

有时需要在 HTTP request 头信息中提供一个 referer 字段, 表示是从哪里跳转过来的。

```
$ curl --referer http://www.example.com http://www.example.com
```

### ➤ User Agent 字段

这个字段用来表示客户端的设备信息。服务器有时会根据这个字段, 针对不同设备返回不同格式的网页, 比如手机版和桌面版。

例如, Firefox 的 UserAgent:

```
Mozilla/5.0 (platform; rv:geckoversion) Gecko/geckotrail Firefox/firefoxversion
```

curl 可以这样模拟:

```
$ curl --user-agent "[User Agent]" [URL]
```

### ➤ Cookie

使用--cookie 参数, 可以让 curl 发送 Cookie。

```
$ curl --cookie "name=xxx" www.example.com
```

至于具体的 Cookie 的值, 可以从 HTTP response 头信息的 Set-Cookie 字段中得到。

另外还可以将 Cookie 暂存为文件反复操作:

- c cookie-file, 可以保存服务器返回的 Cookie 对象并写入本地文件;
- b cookie-file, 可以使用这个本地文件作为 Cookie 信息, 进行后续的请求。

```
$ curl -c cookies http://example.com
```

```
$ curl -b cookies http://example.com
```

### ➤ 增加头信息

有时需要在 HTTP request 中自行增加一个头信息, --header 参数就可以起到这个作用。

```
$ curl --header "Content-Type:application/json" http://example.com
```

### ➤ HTTP认证

有些网域需要 HTTP 认证, 这时 curl 需要用到 --user 参数。

```
$ curl --user name:password example.com
```

以上这些都只是 curl 的一些最基本也是最常用的参数功能, curl 是一个非常强大的命令工具, 如果了解它的全部功能, 则可以用 -h 参数对其逐一了解。本书由于篇幅所限, 对于其他不常用的功能就不一一解释了。

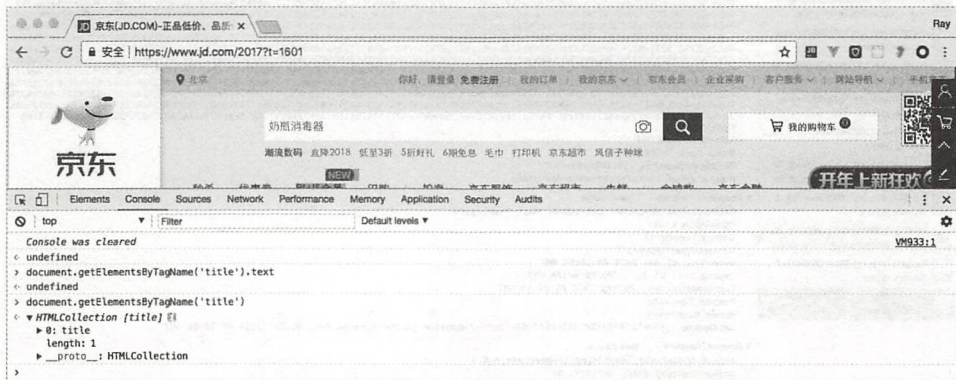
## 浏览器中的开发人员工具

除了 curl, 常用的还有浏览器的开发人员工具 (Developer tools), 这种工具来源于 Firefox 中的 Firebug。后来 Firefox、Chrome、Safari 和 Edge 也陆续将其作为标准功能加入浏览器中。

本节就以 Chrome 为例, 简单介绍一下开发人员工具的作用。Chrome 的开发人员工具的功能比较齐全, 功能也相当强大。在爬虫系统开发中用到它, 大多数是在进行网页元素分析、实质测试一些 DOM 的运行变量, 或者以结构化方式查看 JSON 格式的响应数据。

### ➤ Console

Console 在 Chrome 中的示意图如下图所示。



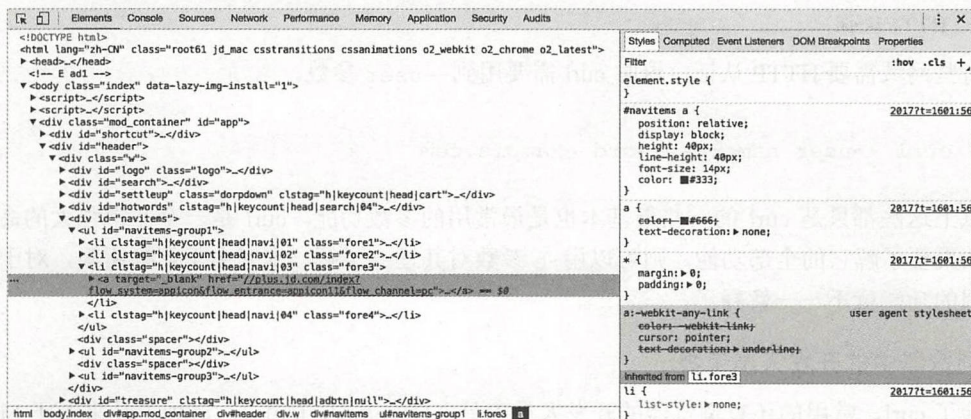
我们可以直接在 Console 中键入 javascript 执行 Console, 这个功能在测试一些 DOM 的 CSS 选择器, 或者简单的 JavaScript 脚本时相当实用。

### ➤ 查看元素结构

Element 页是最常用的, 只要在网页上右键单击某一元素, 单击“检查”就可以直接启动开

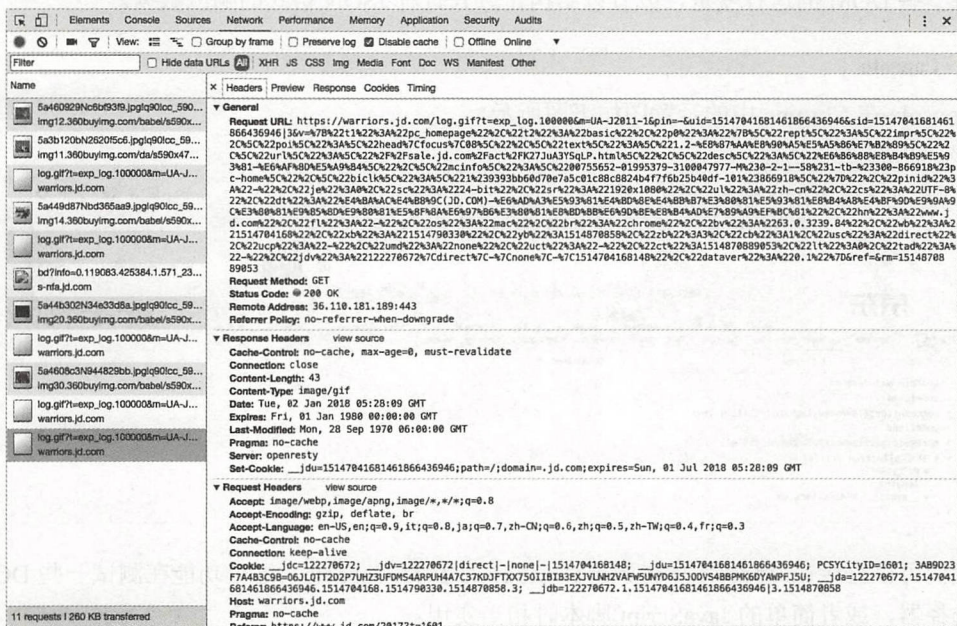


发人员工具并以树状结构定位到该元素上, 对于分析元素的定位和快速查看与其他容器之间的关系显得很有必要, 如下图所示。



## ► 查看请求与响应

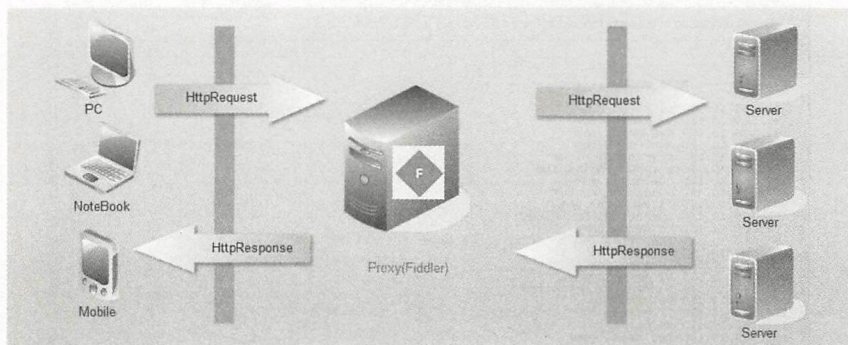
在 Networking 页中可以查看当前网页到底发送过多少的请求, 每个请求、响应的详细情况都可以了如指掌, 如下图所示。



## ► Fiddler

Fiddler (中文名称: 小提琴) 是一个 HTTP 的调试代理, 以代理服务器的方式监听系统的

HTTP 网络数据流动。Fiddler 可以检查所有 HTTP 通信, 设置断点, 以及查看所有的“进出”的数据(一般用来抓包)。Fiddler 还包含一个简单却功能强大的基于 JScript .NET 事件脚本子系统, 它可以支持众多的 HTTP 调试任务。Fiddle 协议示意图如下图所示。

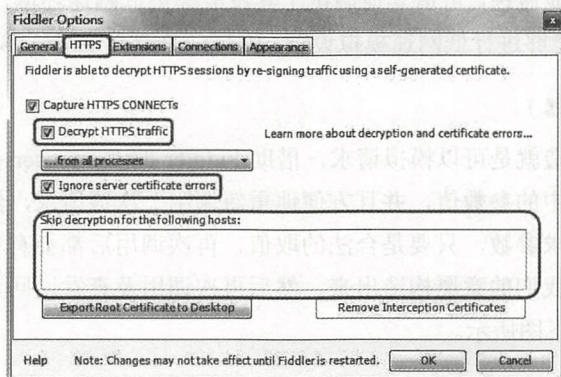


Fiddler 是以代理 Web 服务器的形式工作的, 浏览器与服务器之间通过建立 TCP 连接以 HTTP 协议进行通信。浏览器默认通过自己发送 HTTP 请求到服务器, 它使用代理地址: 127.0.0.1; 端口: 8888。当 Fiddler 开启后会自动设置代理, 退出时会自动注销代理, 这样就不会影响别的程序。如果 Fiddler 非正常退出, 这时因为 Fiddler 没有自动注销, 则会造成网页无法访问。解决的办法是重新启动 Fiddler。

如果系统正在使用代理, 那么要先将代理关闭, 否则会无法正常访问。

### ► 监听 HTTPS

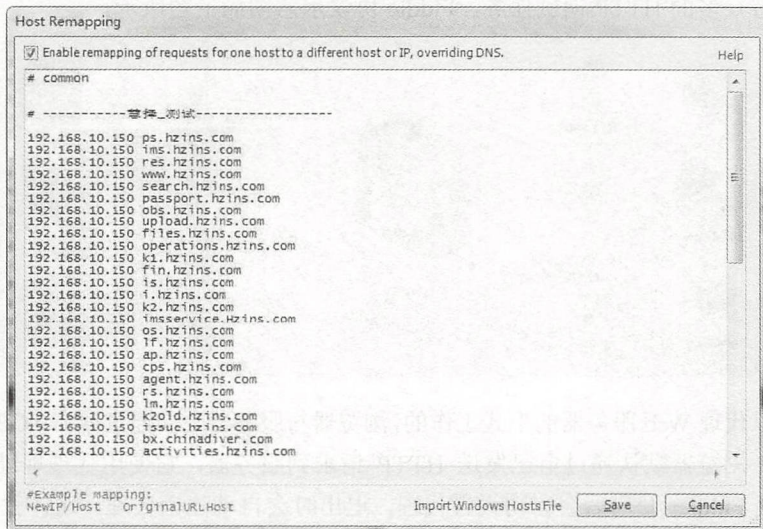
Fiddler 不仅能监听 HTTP 请求, 而且默认情况下也能捕获 HTTPS 请求, 在 Tool→Fiddler Option→HTTPS 下进行设置, 勾选“Decrypt HTTPS traffic”。如果不必监听服务器端的证书错误, 则可以勾选“Ignore server certification errors”, 也可以跳过几个指定的 HOST 来缩小或者扩大监听范围, 如下图所示。





### ➤ Host切换

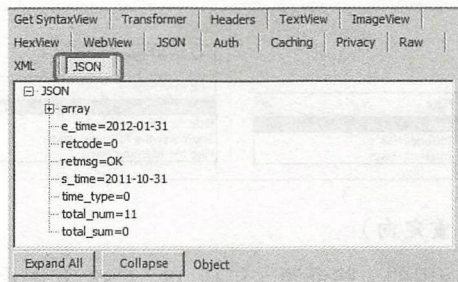
也就是更改本机上的 Host 文件中的地址映射表, 如下图所示。



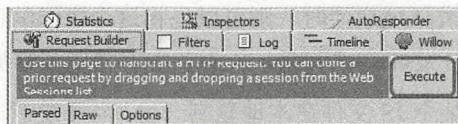
- 模拟各类场景;
- 通过 GZIP 压缩, 测试性能;
- 模拟 Agent 测试, 查看服务端是否对不同客户端定制响应;
- 模拟慢速网络, 测试页面的容错性;
- 禁用缓存, 方便调试一些静态文件或测试服务端响应情况;
- 根据一些场景自定义规则;
- 有时出于兼容性考虑或者对某处进行性能优化, 在低网速下往往能较快发现问题所在, 也容易发现性能瓶颈, 可惜其他调试工具没能提供低网速环境, 而强大的 Fiddler 考虑到了这一点, 能够进行低网速模拟设置: Rules→Performance→Stimulate Modem Speeds。

### ➤ Composer (构造器)

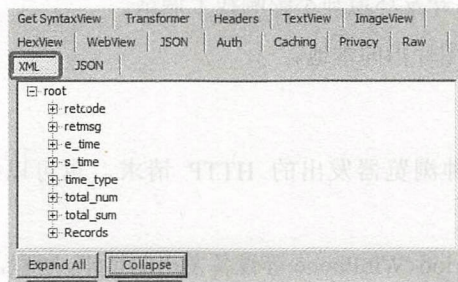
顾名思义, 请求构造就是可以模拟请求, 借助 Fiddler 的 Composer 在不改动开发环境实际代码的情况下修改请求中的参数值, 并且方便地重新调用一次该请求, 然后比较两次请求响应有何不同。任何一个请求参数, 只要是合法的取值, 再次调用后都会有相应的响应。任意一个合法请求组合能够按照我们的意愿构造出来, 然后再次调用及查看返回数据。例如, 查看请求返回的 JSON 数据, 如下图所示。



将该请求用鼠标左键单击拖入 Fiddler 右侧 Request Builder 标签内并修改原请求参数 OutPutType=JSON 为 OutPutType=XML, 然后单击 Execute 按钮再次触发调用请求, 如下图所示。

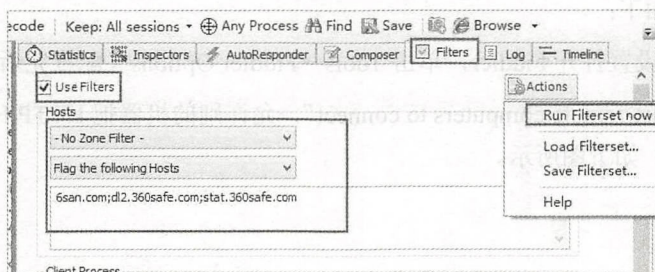


双击这次请求包, 在 Inspectors 标签下查看到返回的数据为 XML 格式, 而 JSON 格式一栏为空, 如下图所示。

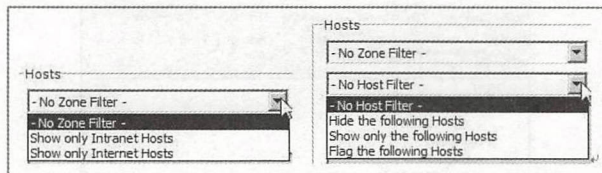


### ► Filters (过滤监控)

对一个重新载入的页面进行抓包, 如果包的条目过多而需要关注的就那么几项, 则可以使用 Fiddler 的过滤器 Filters 进行抓包, 抓包时只会抓取希望抓到的那些包。切换到 Filters 标签, 勾选 Use Filter, 以便激活过滤器, 这样就可以选择下面各种过滤方式了, 如下图所示。







### ➤ AutoResponder (请求重定向)

所谓请求无非就是需要调用的一些资源（包括 JS、CSS 和图片等），重定向就是将页面原本需要调用的资源指向其他资源（能够控制的资源或者可以引用的资源）。

- 将前台服务器某个或多个资源在本地做个副本，如果在正常网络访问环境下该资源出现了 Bug 而导致开发环境崩溃时，则可以先将这个资源的请求重定向到本地副本，这样就可以继续进行开发并调试页面，从而节省大量资源维护的等待时间。
- 将多人同时维护的某个 JS 文件在本地复制一份，当开发调试受到他人调试代码干扰时，可以将这个 JS 的调用重定向到本地无干扰的 JS 文件，进行无干扰开发，功能开发完成并调试好之后再将代码合入开发环境中，这样就可以避免受到他人干扰。也就是说，能够将 JS 文件脱离开发环境却不影响线上调试。

还可以将样式文件或者图片指向本地。

### ➤ 移动端抓包

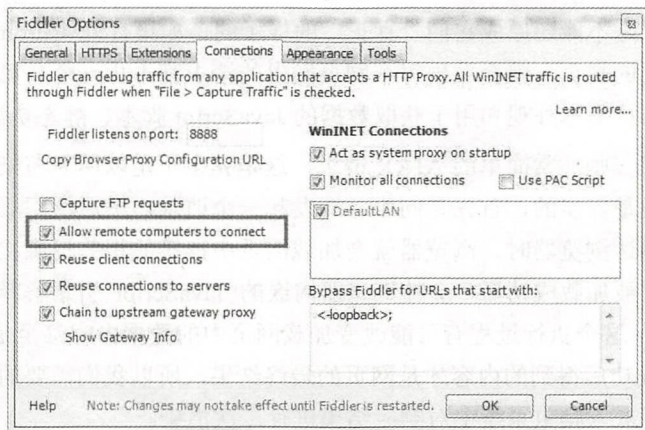
Fiddler 不但能截获各种浏览器发出的 HTTP 请求，也可以截获各种智能手机发出的 HTTP/HTTPS 请求。

Fiddler 能捕获 iOS、Andriod、WinPhone 等设备发出的请求，同理，也可以截获 iPad、MacBook 等设备发出的 HTTP/HTTPS。

注意，安装 Fiddler 的机器要和 iPhone 在同一个网络中，否则 iPhone 不能把 HTTP 请求发送到 Fiddler 的机器上。

具体操作步骤如下：

- 在 Fiddler 设置打开 Fiddler，单击 Tools→Fiddler Options（配置完后要重启 Fiddler）。
- 选中“Allow remote computers to connect”，允许别的机器把 HTTP/HTTPS 请求发送到 Fiddler 上，如下图所示。



完成这一步后,在 iPhone 或者 Android 的网络设置中将代理服务器主机名改为 Fiddler 所在的机器名,代理端口也需要设置。这样所有由手机发出的请求都会被 Fiddler 截获。

在 Mac 下运行 Fiddler 只能使用 mono 的 32 位版本。

mono --arch=32 Fiddler.exe。

使用 Fiddler 可以解决很多我们在分析、设计爬虫系统时,由于对网络后面传输的具体内容不清楚而导致的各种问题。它可谓是黑暗中最明亮的眼睛。

在 Windows 平台上使用 Fiddler 会比较流畅,而 macOS 上其性能就相当低下,使用体验非常差。可能是由于 Fiddler 在本章所介绍的工具中属于功能极为强大的一类,大而全的工具往往都难以做到尽善尽美,但在某些场景下它仍然占据了重要的地位。

## 4.5 处理JavaScript

近几年随着前端技术和手机端 App 的飞速发展,互联网架构也发生了天翻地覆的改变。尤其是 Angular、React 和 Vue 这类优秀且强大的前端框架的大面积应用,过去基于纯后端的 Web 结构已经明显过时了。你会发现大量网站尤其是那些互动性特别强的网站几乎都是采用前后端分离架构的。

所谓前后端分离就是指一个网站由前端与后端两拨人独立开发,相互之间只通过 JSON 作为通信接口,从开发到部署几乎可以没有交集。可以说这是一种互联网开发的垂直化的做法。其实即使没有完全采用前后分离结构的网站,在现代的前端发展节奏下,JavaScript 化的程度也是越来越高,这一点可以追溯到 jQuery 诞生的时代。

这样的网站对于爬虫来说无疑是一种灾难,因为脚本化的网页如果没有脚本引擎在客户端



进行二次解释执行，是不能还原成它们“真正”的样子的。如果打开 Chrome 的开发者模式，查看这类重度前端化的网页，就会发现每个网页发出的请求都不止一个！第一次请求下载的通常都只是用于渲染网页基本外观和用于获取数据的 JavaScript 脚本、静态资源或样式表等。

Scrapy 只能处理一些非常简单的 AJAX 请求，这是完全不足以用于渲染充满复杂性的前端网页的。网页的加载是异步的，首先是向服务端发起一个请求，然后返回服务器解释后的文本结果，当这个结果到达浏览器时，浏览器就会加载网页中链接的相关资源文件，如脚本与样式表。当脚本与样式表被加载成功后，会被浏览器内嵌的 JavaScript 引擎解释执行（在浏览器的 onload 事件中执行），这个执行过程有可能改变加载网页中的数据内容甚至是元素（DOM）。等待 JavaScript 执行成功后得到的内容才是网页的最终结果，所以我们需要的是一个可编程控制的浏览器，这样才能嵌入爬虫系统中对响应结果进行二次渲染。

要在 Python 中处理 JavaScript 网页有两种办法：

(1) Selenium + WebDriver。

(2) Splash。

它们主要应用于软件的自动化外部测试，所有它们会有非常多关于浏览器控制与网页互操作的相关内容。对于爬虫系统而言，很多内容是不需要的，爬虫只需要一个能执行 JavaScript 渲染的工具包即可。因此在介绍它们的用法时，不会对 Selenium 和 Splash 的内容进行全面讲述，而是侧重于它们在爬虫系统中最典型的用法。

那什么内容才是最典型的应用呢？主要有以下几种：

(1) 单向导航。

(2) 执行脚本。

(3) 读取响应内容正文与定位元素。

(4) 处理 Cookies。

明确了以上几点后就不会在 Selenium 和 Splash 庞大的文档中迷失，因为很多内容即使学了，在爬虫系统中也未必用得上。

接下来就分别讲一下 Selenium 和 Splash 是如何在爬虫系统中使用的。

### 4.5.1 示例：电商产品爬虫

为了能突出 Selenium 与 Splash 之间的差别，在讲述它们之前先引入一个示例，以示例的实际需求入手会更容易理解爬取 JavaScript 网页的具体思路与方法，同时也更清楚在不同的场景中如何在 Selenium 与 Splash 之间做出选择。

## 示例背景

这个示例是从我的团队所负责的一个项目中提炼出来的，由于我所在的公司有一部分业务是做电子商务（以下简称电商）的运营，主要是代理一些知名运动品牌的产品。电商在中国是如此的兴旺，可想而知其竞争之激烈。作为一家电商企业，实际的收益来源就是从消费者与厂家之间赚取差价。在相同的销售量下，差价越大自然代表着收益率就越高。只要上淘宝搜索某一标准产品的货号，例如：匡威的 150154c 这一款鞋，就会看到同一款鞋有各种各样的价格，消费者一般会选价格最低销量最高的一家来购买。那么问题来了：

在厂家给出的价格与淘宝上销售的同款最高价格之间应该如何进行科学的定价？

有经验的操盘手可能会马上给出一个他（她）认为的合理价位，但这可能只能代表某个人的主观建议，在大数据面前往往是苍白无力的。

那数据如何获得呢？最简单的办法是上淘宝的卖家工具那里买。但电商运营企业又不会单单在某一个电商平台上进行销售，如果要同时在淘宝、天猫、京东、唯品会这些平台销售同一款产品，那定价依据又如何得出呢？我所见到的“经典”做法就是在各个平台都购买卖家工具，每个卖家工具都可以将自己的数据导出成 CSV 或者 Excel 文件，然后通过人工方式合成后进行统计。一份单品的数据分析统计报告就得有专人处理两三天，那一个行业级的统计报告要处理多久呢？

这正是这个示例的一个小背景，也是真实存在的场景。在此背景下，这个示例假设我们有一份产品货号清单，只要在淘宝的搜索框中输入某一个货号就可以得到这个产品在售的所有店铺的价格，我们需要爬虫将前 10 页的产品价格与店铺都收集下来，作为后台数据的分析依据，而且这个爬虫是每天不定期地到网站上爬取一次，以更新当天的价格数据。

## 设计思路

许多电商网站是有很强大的反爬机制的，技术壁垒与爬取成本也可以成为反爬机制的一种外围防范手段。对于爬虫系统而言，第一技术壁垒就是 JavaScript 网页，因为 JavaScript 是对网页进行异步渲染的。也就是说，第一次发出的请求后得到的响应可能只是一堆的 JavaScript 与静态资源的链接，这个页面需要浏览器中的 JavaScript 引擎在客户端中执行一次才会展现出其真正的样子。

网页被正确地渲染之后所做的数据分析与提取工作都是相同的，得到的数据结构也是一样。那么，我们可以忽略 JavaScript 的处理复杂性，只将其当作“间接性”获取网页的手段，先分析数据 Item 的结构与目标结构的网页元素结构，最后分别将 Selenium 和 Splash 接入爬取的部分中。

首先来看一下产品搜索页面，如下图所示。





从此页面我们就已经可以得到几个爬虫需要知道的基本要素:

- (1) URL: 某电商网站 URL, 如 `http://s.taobao.com`。
- (2) `allowed_domains`: `s.taobao.com`。
- (3) 产品 Item 的基本结构。

接下来可以快速定义产品 Item 对象, 代码如下所示。

```
from scrapy import Item, Field

class ProductItem(Item):
    name = Field()           # 品名
    link = Field()           # 链接地址
    sn = Field()             # 货号
    image_url = Field()      # 产品图片地址
    image_path = Field()     # 图片下载至本地的位置
    price = Field()          # 价格
    deal = Field()           # 成交人数
    free_shipping = Field()  # 是否包邮
    shop = Field()           # 淘宝店名
    location = Field()       # 地区
```



将页面拉到最下方还有一个分页，这可不能忘记，因为这是爬虫进行跟进的重要依据。



然而，对于统计样本的采集一般只需要 28 到 36 个就具有统计意义了，因此我们并不需要将所有符合条件的数据都全拉下来，只要前 36 项就足够了。这一点很重要，不是说爬虫爬的东西越多越好，而是越精确越好。

依据以上分析与规则，按前文介绍的爬虫写法（暂且将产品搜索页看作不是 JavaScript 生成的），先构建一个蜘蛛的基本结构。

以下是本示例中采用的产品货号，保存于 product\_data.py 文件中：

```
product_sns = [
    '101001',
    '101013'
    '158369C',
    '150268',
    '142334c',
    '156733c',
    # ...
]
```

以下为蜘蛛的部分实现代码：

```
# coding:utf-8
from scrapy import Request, Spider
from urllib.parse import quote
from .items import ProductItem
from product_data import product_sns # 导入货号

class TaobaoItemSpider(Spider):
```







```

name = 'taobao'
allowed_domains = ['www.xxx.com']
base_url = 'https://s.xxx.com/search?q=%s'

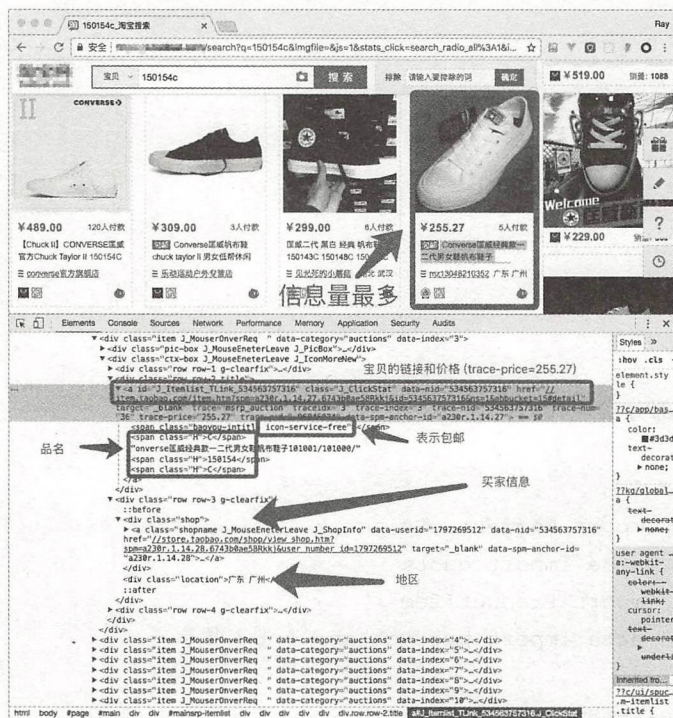
def start_requests(self):
    for sn in product_sns:
        keyword = u'匡威%s' % sn
        url = base_url % keyword
        yield Request(url=url, callback=self.parse, meta['sn']=sn)

def parse(self, response):
    pass

```

首先定义一个 `base_url`，即商品列表的 URL，其后拼接一个搜索关键字，就是该关键字在淘宝的搜索结果商品列表页面。

接下来用 Chrome Developer Tools 分析元素结构，网页结构是相当明确的，具体分析结果如下图所示。





然后就可以实现 parse 方法了:

```
class TaobaoItemSpider(Spider):
    # ... 省略

    def parse(self, response):

        products = response.css('#main-srp-itemlist .items .item')

        for product in products:
            item = ProductItem()
            item['price'] = product.css('.price>strong::text').extract_first()
            item['name'] = ''.join(product.css('div.title>a::text').extract()).
strip()
            item['shop'] = ''.join(product.css(".shopname>span::text").
extract()).strip()
            item['image_url'] = product.css('.pic img::attr(data-src)').
extract_first().strip()
            item['deal'] = product.css('.deal-cnt::text').extract_first()
            item['location'] = product.css('.location::text').extract_first()
            item['sn'] = response.meta['sn']
            item['link'] = product.css('div.title>a::attr(href)').extract_
first()
            item['free_shipping'] = product.css(".icon-service-free").
extract_first() != None

            yield item
```

现在整个蜘蛛的基本结构已经完成, 接下来进入本节的主题, 将以上示例接入 JavaScript 解释引擎使其能真正运行起来。

## 4.5.2 Selenium和PhantomJS

Selenium 是一个基于 Java 开发的自动化浏览器处理器, 从另一个角度来说, 它更像是一个浏览器驱动的代理。它提供了一套访问浏览器内核的编程接口, 程序可以调用 Selenium 的编程接口以控制浏览器的行为, 甚至操控 DOM。它自身是没有配备浏览器的, 因此需要配合本机上安装的浏览器驱动一同使用, 例如, Firefox、Chrome、Safari 等。由于自动化测试与持续集成的







大面积推广, 使得 Selenium 被广泛地应用于各种语言平台。Node.js、Python、Ruby、Java 等主流 Web 开发语言工具链中必然会发现它的身影。

Selenium 支持各种主流浏览器, 更重要的是, 它的接口是统一的, 并不需要因为更换浏览器而改动任何编码。因此我们也可以使用它来作为爬虫渲染动态网页的工具。

### 安装Selenium

可以从 PyPI 网站 (<https://pypi.python.org/simple/selenium/>) 下载 Selenium 库, 也可以通过第三方管理器 (像 pip) 使用命令行进行安装:

```
$ pip install selenium
```

Selenium server 是一个 Java 程序, 推荐使用 1.6 及以上的 JRE 来运行 Selenium server。可以从这里 (<https://docs.seleniumhq.org/download/>) 下载 2.x 的 Selenium server, 文件名看起来应该类似于 selenium-server-standalone-2.x.x.jar, 任何时间都可以下载到最新的 2.x Selenium server。

如果机器上没有安装 JRE, 则可以从 JRE from the Oracle (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>) website 下载。如果使用的是 Linux 系统并且有 ROOT 权限, 也可以使用系统命令来安装 JRE。如果 PATH (环境变量) 中 Java 命令是可用的, 则可以用这个命令来开启 Selenium server:

```
$ java -jar selenium-server-standalone-2.x.x.jar
```

把 2.x.x 替换成从网上下载的版本。如果是一个非 ROOT 用户安装的 JRE, 或者环境变量里 java 命令不可用, 输入 java 命令的相对路径或者绝对路径, 则同样可以提供 Selenium server 的相对路径或者绝对路径:

```
/path/to/java -jar /path/to/selenium-server-standalone-2.x.x.jar
```

### PhantomJS

如果直接采用 Firefox、Chrome、Safari 这类带有界面的浏览器, 则会带来严重的问题。因为 Selenium 会启动一个专用的进程来加载这些浏览器, 也就是说, Selenium 会自动引导本机上安装的浏览器, 如果是上述那些带有界面的正常浏览器, 则会看到浏览器像平常那样打开并显示出来, 不过这样做对系统的负担相当大, 界面的加载要消耗相当大的系统资源。Chrome 或者 Edge 打开一个浏览器窗口就会消耗 200MB~300MB 的内存, 一旦爬虫系统启用多个实例并行爬区任务, 系统将不堪重负, 而解决这个问题的办法就是将 PhantomJS 作为浏览器。

PhantomJS 是一个“无头”(headless)浏览器, 使用著名的 V8 引擎构建。它会把网站加载





到内存中并执行页面上的 JavaScript，但不会向用户展示网页的图形界面。将 Selenium 和 PhantomJS 结合在一起，就可以运行一个非常强大的网络爬虫了。虽然没有界面，但 DOM 渲染、JS 运行、网络访问、Canvas/SVG 绘制等功能都很完备，在页面抓取、页面输出、自动化测试等方面有着广泛的应用。

Selenium+PhantomJS 可以说是 Selenium 中最实用也是被应用得最为普遍的方案。PhantomJS 可以看作没有人机交互界面的全功能型的 Chrome。由于没有人机交互界面和大量内存的损耗，其具有极高的运行效率与速度，因此它将是接下来作为爬虫渲染动态网页的方案之一。

将上面的代码运行一次，记录下运行所需的时间，然后换成 PhantomJS 进行对比，其速度的差异会让你做出正确的选择：

```
from selenium import webdriver

browser = webdriver.PhantomJS()    # 加载 PhantomJS 浏览器
browser.get("http://www.xxx.com")  # 打开某电商网站
browser.implicitly_wait(1)         # 等待 1 秒
print browser.get_cookies()
```

### 安装 PhantomJS

PhantomJS 是一个跨平台的开源项目，因此它提供了各个平台的独立安装包，可以到 PhantomJS 的官方下载页 (<http://phantomjs.org/download.html>) 上获取。PhantomJS 官方提供了两种安装方式，一种是下载 PhantomJS 的压缩包后在本地解压安装，另一种则是直接下载源码 (<http://www.github.com/ariya/phantomjs>)，在本地机上构建，方法比较简单，在此就不赘述了。

### 简单使用

如果已经安装好 Python 和 Selenium，则可以这样开始使用：

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

driver = webdriver.Firefox()
driver.get("http://www.python.org")
assert "Python" in driver.title
elem = driver.find_element_by_name("q")
elem.send_keys("pycon")
elem.send_keys(Keys.RETURN)
assert "No results found." not in driver.page_source
```







```
driver.close()
```

把上面的脚本保存到文件（例如，python\_org\_search.py）中，然后就可以运行它了：

```
$ python python_org_search.py
```

selenium.webdriver 模块提供了所有 WebDriver 的实现，现在支持 WebDriver 的实现的有 Firefox、IE、Chrome、Remote、Keys 类提供了键盘的代码（回车、ALT、F1 等）。

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
```

然后创建一个 Firefox 的实例：

```
driver = webdriver.Firefox()
```

driver.get 方法会导向给定的 URL 的页面，WebDriver 会等待页面完全加载完（就是 onload 函数被触发了）才把程序的控制权交给测试或者脚本。如果页面用了太多的 AJAX，那么这个机制就没什么用了，因为它不知道页面到底是什么时候加载完。

```
driver.get("http://www.python.org")
```

WebDrive 提供了一系统类似于 find\_element\_by\_\* 的方法来寻找页面元素，例如，我们利用 find\_element\_by\_name 方法，通过元素的 name 属性来定位一个文本输入框元素。更详细的寻找元素的方法可以参阅本章关于元素定位的内容。

```
elem = driver.find_element_by_name("q")
```

接着我们发送了一些字符，类似于用键盘直接输入。我们可以导入 selenium.webdriver.common.keys，然后用 Keys 类来表示特殊的键盘符：

```
elem.send_keys("pycon")
elem.send_keys(Keys.RETURN)
```

最后浏览器窗口被关闭了，也可以调用 quit 方法来代替 close，区别在于 quit 会退出整个浏览器，而 close 只会关闭一个标签。如果浏览器只有一个标签，那么这两个方法完全一样，都会关闭整个浏览器。





```
driver.close()
```

由于 Selenium 和 PhantomJS 有很多技术内容,要熟练地使用 Selenium 和 PhantomJS 编写爬虫,需要先对 Selenium 的基本使用有所了解,接下来就从五个方面对 Selenium 的使用进行介绍。

### 4.5.2.1 导航

用 WebDriver 要做的第一件事就是指定一个链接,一般使用 `get` 方法:

```
driver.get("https://www.taobao.com")
```

WebDriver 会等待页面完全加载完(就是 `onload` 函数被触发了)才交还程序的控制权,这种加载方式称为阻塞式加载。如果页面用了太多的 AJAX,那么这个机制就没什么用了,因为它不知道页面到底什么时候加载完。可以使用 `waits` 来确定页面是否完全加载完了。

#### 页面交互

我们比较喜欢做的事情就是和页面交互,准确地说,是和页面里的 HTML 元素交互。首先,我们要找到一个元素,WebDriver 提供了许多方法查找元素,例如,给定一个这样的元素:

```
<input type="text" name="passwd" id="passwd-id" />
```

可以用下列任意方法找到它:

```
element = driver.find_element_by_id("passwd-id")
element = driver.find_element_by_name("passwd")
element = driver.find_element_by_xpath("//input[@id='passwd-id']")
```

也可以通过文本信息来找到一个链接,但是要注意,文本必须要完全匹配。在使用 XPath 时也要注意,如果有多个元素匹配,则只会返回第一个。如果匹配不到任何元素,则会抛出一个 `NoSuchElementException` 异常。

WebDriver 有一个基于对象的 API,我们可以通过同一个接口代表所有类型的元素,这意味着当敲击 IDE 的自动补全组合键时,虽然可以调用很多方法,但不是所有方法都行得通。不过不要担心,WebDriver 会自己尝试做正确的选择。如果调用一个没用的方法(例如,在一个 `meta` 标签上调用 `setSelected()`),则 WebDriver 会抛出一个异常。

当获取到一个元素后,我们可以做些什么呢?首先,可能输入一些文本到一个文本区域:







```
element.send_keys("some text")
```

可以使用 `Keys` 类来模拟输入方向键:

```
element.send_keys(" and some", Keys.ARROW_DOWN)
```

理论上任意元素都可以调用 `send_keys` 方法,也就是说,我们可以测试如 Gmail 的键盘快捷键。`send_keys` 的副作用就是输入文本到文本域不会自动清除,而是会附加到原有的文本后面,我们可以使用 `clear` 方法来清除文本框或者文本域的内容。

```
element.clear()
```

对于爬虫而言,这种完全模仿人类行为的交互在一般的爬虫中用得比较少,要交互就需要浏览器的 JavaScript 对行为进行解析,这样做会比较低效。交互的目的是通过友好的人机界面获得一个新的请求地址,那么在爬虫中直接生成这个请求地址就好了。

## 填充表单

我们已经知道怎么向一个文本框和文本域输入内容,但是其他元素要怎么处理呢?可以触发下拉选框,并且用 `setSelected` 方法来让一个选项被选中,处理选择框不会很困难:

```
element = driver.find_element_by_xpath("//select[@name='name']")
all_options = element.find_elements_by_tag_name("option")
for option in all_options:
    print("Value is: %s" % option.get_attribute("value"))
    option.click()
```

这段代码会找到页面的第一个选择框元素,然后遍历每个选项,输出它们的值,并且依次选中。

可以看到,这种方式处理选择框不太高效,WebDriver 支持许多类,其中包括一个 `Select` 的类,给我们提供了许多有用的方法:

```
from selenium.webdriver.support.ui import Select
select = Select(driver.find_element_by_name('name'))
select.select_by_index(index)
select.select_by_visible_text("text")
select.select_by_value(value)
```





WebDriver 也提供了取消选中选项的方法：

```
select = Select(driver.find_element_by_id('id'))
select.deselect_all()
```

上面的代码会取消页面第一个选择框的所有选中项。

假设测试中我们需要所有默认选中项的列表，Select 类提供了一个属性：

```
select = Select(driver.find_element_by_xpath("xpath"))
all_selected_options = select.all_selected_options
```

获取所有可用的选项：

```
options = select.options
```

一旦填写完表单，一般就要提交表单，一个方法是找到提交按钮，然后单击它：

```
# Assume the button has the ID "submit" :)
driver.find_element_by_id("submit").click()
```

WebDriver 对每个元素都提供了一个 submit 方法，如果在一个表单中的元素上调用，则 WebDriver 会沿着 DOM 树往上一直寻找，直到找到一个闭合的表单为止，然后调用 submit 方法；如果元素不在表单中，则会抛出一个 NoSuchElementException 异常。

## 拖放

可以使用拖放功能移动确定数量的元素，或者拖到另一个元素上面：

```
element = driver.find_element_by_name("source")
target = driver.find_element_by_name("target")

from selenium.webdriver import ActionChains
action_chains = ActionChains(driver)
action_chains.drag_and_drop(element, target).perform()
```

## 在窗口（window）和框架（frame）间移动

现在的网页应用中没有页面框架或者只用一个窗口就包含所有内容的情况已经很少了。

WebDriver 支持在指定的窗口间移动，方法为 switch\_to\_window：





```
driver.switch_to_window("windowName")
```

现在所有 driver 的调用都会指向这个给定的窗口，但是我们如何知道窗口的名字是什么呢？可以打开这个窗口的 JavaScript 脚本或者 link 链接：

```
<a href="somewhere.html" target="windowName">Click here to open a new window</a>
```

或者，可以传一个 window handle 给 switch\_to\_window() 方法，它可以迭代每一个打开的窗口：

```
for handle in driver.window_handles:  
    driver.switch_to_window(handle)
```

也可以在框架和框架之间进行切换：

```
driver.switch_to_frame("frameName")
```

我们可以用 . 分离路径来访问子框架，并且可以指定它的索引：

```
driver.switch_to_frame("frameName.0.child")
```

这会跳到 frameName 框架中第一个名为 child 的子框架。一旦操作完了框架，就可以通过下面的操作回到父框架：

```
driver.switch_to_default_content()
```

### 导航：历史记录和定位

可以使用 get 命令（driver.get("http://www.example.com")）导航到一个页面。WebDriver 有一些较小的、侧重任务的接口，导航是一个很有用的任务，要打开一个页面，可以使用 get 方法：

```
driver.get("http://www.example.com")
```

在浏览器的历史记录中后退或者前进：

```
driver.forward()
```

```
driver.back()
```

这些函数完全依赖于底层驱动，如果过去习惯某一个浏览器的运行状态，当切换到新的浏览器时，调用这些方法有可能会出现预料之外的情况。

#### 4.5.2.2 元素定位

有许多方法可以对页面的元素进行定位，可以根据自己的需要选择最合适的一种。Selenium 提供了下面的方法来进行元素定位：

- `find_element_by_id`
- `find_element_by_name`
- `find_element_by_xpath`
- `find_element_by_link_text`
- `find_element_by_partial_link_text`
- `find_element_by_tag_name`
- `find_element_by_class_name`
- `find_element_by_css_selector`

寻找多个元素（下列方法会返回一个 list，其余使用方式相同）：

- `find_elements_by_name`
- `find_elements_by_xpath`
- `find_elements_by_link_text`
- `find_elements_by_partial_link_text`
- `find_elements_by_tag_name`
- `find_elements_by_class_name`
- `find_elements_by_css_selector`

除了上面这些公有的方法，还有 2 个私有的方法来实现页面对象的定位。这两个方法就是 `find_element` 和 `find_elements`：

```
from selenium.webdriver.common.by import By
```

```
driver.find_element(By.XPATH, '//button[text()="Some Text"]')
driver.find_elements(By.XPATH, '//button')
```



By 类的可用属性如下:

```
ID = "id"
XPATH = "xpath"
LINK_TEXT = "link text"
PARTIAL_LINK_TEXT = "partial link text"
NAME = "name"
TAG_NAME = "tag_name"
CLASS_NAME = "class name"
CSS_SELECTOR = "css selector"
```

### 根据id定位

如果知道元素的 id 属性, 则可以使用 id 进行定位。在 id 定位中, 会返回第一个 id 属性匹配的元素, 如果没有元素匹配, 则抛出 NoSuchElementException 异常。

举个例子, 我们来看一个页面:

```
<html>
<body>
  <form id="loginForm">
    <input name="username" type="text" />
    <input name="password" type="password" />
    <input name="continue" type="submit" value="Login" />
  </form>
</body>
</html>
```

可以这样定位表单元素 form:

```
login_form = driver.find_element_by_id('loginForm')
```

### 根据name定位

如果知道元素的 name 属性, 则可以使用 name 进行定位。在 name 定位中, 会返回第一个 name 属性匹配的元素, 如果没有元素匹配, 则抛出 NoSuchElementException 异常。

我们再来看一个页面:

```
<html>
<body>
```

```
<form id="loginForm">
  <input name="username" type="text" />
  <input name="password" type="password" />
  <input name="continue" type="submit" value="Login" />
  <input name="continue" type="button" value="Clear" />
</form>
</body>
<html>
```

username 和 password 元素可以这样定位：

```
username = driver.find_element_by_name('username')
password = driver.find_element_by_name('password')
```

下面这个操作会返回 Login 按钮，因为它在 Clear 按钮的前面：

```
continue = driver.find_element_by_name('continue')
```

## XPath定位

XPath 是用来定位 XML 文档节点的语言。不过 HTML 可以看作 XML (XHTML) 的一种实现。Selenium 用户可以使用这个强力的语言来瞄准 Web 应用的元素。XPath 延伸了用 id 或者 name 属性来定位的单一方法，开创了许多可能性，例如，定位页面的第三个复选框。

用 XPath 的主要理由之一，就是想定位的元素没有合适的 id 或者 name 属性时，可以用 XPath 来对元素进行绝对定位（不推荐），或者把这个元素和另外一个有确定 id 或者 name 的元素关联起来（即相对定位）。XPath 定位器也可以用来找出那些具有 id、name 以外属性的元素。

绝对的 XPath 定位包含了从 HTML 根节点起的所有元素，并且一些轻微的改变就会失效。而用 id 或者 name 属性来找到一个靠近的元素（比较理想的是父元素），就可以依靠它们的相对关系来确定目标元素的位置。这种情况改变的可能性就小了很多，我们写的测试程序也会更加可靠。

再来看一个实例：

```
<html>
<body>
  <form id="loginForm">
    <input name="username" type="text" />
    <input name="password" type="password" />
    <input name="continue" type="submit" value="Login" />
```



```

        <input name="continue" type="button" value="Clear" />
    </form>
</body>
<html>

```

form 元素可以这样定位:

```

login_form = driver.find_element_by_xpath("/html/body/form[1]")
login_form = driver.find_element_by_xpath("//form[1]")
login_form = driver.find_element_by_xpath("//form[@id='loginForm']")

```

绝对路径 (如果 HTML 有细微的改变就会失效) HTML 的第一个 form 元素 id 属性为 loginForm 的 form 元素。

username 元素可以这样定位:

```
username = driver.find_element_by_xpath("//form[input/@name='username']")
```

第一个 form 元素的 name 属性是 username。

或

```
username = driver.find_element_by_xpath("//form[@id='loginForm']/input[1]")
```

文档中 id 属性为 loginForm 的第二个 input 子元素。

又或:

```
username = driver.find_element_by_xpath("//input[@name='username']")
```

文档中第一个 name 属性为 username 的 input 子元素。

第一个 input 元素 clear 按钮可以这样定位:

```

clear_button = driver.find_element_by_xpath("//input[@name='continue']
[ @type='button' ]")
clear_button = driver.find_element_by_xpath("//form[@id='loginForm']/
input[4]")

```

type 属性为 button, name 属性为 continue 的第一个 input 元素, id 为 loginForm 的表单的第四个 input 子元素。

### 用链接文本定位超链接

如果知道一个锚标签使用了什么文本，那么就可以使用链接文本定位超链接。在超链接定位中，会返回第一个文本属性匹配的链接，如果没有元素匹配，则抛出 `NoSuchElementException` 异常。

实例：

```
<html>
<body>
  <p>Are you sure you want to do this?</p>
  <a href="continue.html">Continue</a>
  <a href="cancel.html">Cancel</a>
</body>
</html>
```

可以这样定位 `continue.html` 链接：

```
continue_link = driver.find_element_by_link_text('Continue')
continue_link = driver.find_element_by_partial_link_text('Conti')
```

**注：**`find_element_by_partial_link_text` 使用的是子串匹配，只要输入字符串即可匹配。

### 标签名定位

知道元素标签名就可以使用标签名定位，如果没有元素匹配，则抛出 `NoSuchElementException` 异常。

```
<html>
<body>
  <h1>Welcome</h1>
  <p>Site content goes here.</p>
</body>
</html>
```

可以这样定位标题元素 (`h1`)：

```
heading1 = driver.find_element_by_tag_name('h1')
```



### class定位

知道 class 就可以使用 class 定位, 只返回匹配的第一个, 无元素匹配, 会抛出 NoSuchElementException 异常。

实例:

```
<html>
<body>
  <p class="content">Site content goes here.</p>
</body>
</html>
```

定位 p 元素:

```
content = driver.find_element_by_class_name('content')
```

### CSS选择器定位

如果能用 CSS 选择器的语法来表述一个元素, 那么就可以进行 CSS 选择器定位, 只返回匹配的第一个, 无元素匹配, 会抛出 NoSuchElementException 异常。

实例:

```
<html>
<body>
  <p class="content">Site content goes here.</p>
</body>
</html>
```

定位 p 元素:

```
content = driver.find_element_by_css_selector('p.content')
```

#### 4.5.2.3 等待事件

现在很多 Web 应用都在使用 AJAX 技术。浏览器载入一个页面时, 页面内的元素可能是在不同的时间载入的, 这会加大定位元素的困难程度, 因为元素不在 DOM 中, 会抛出 ElementNotVisibleException 异常。使用 waits 就可以解决这个问题。Waiting 给 (页面) 动作的执行提供了一些时间间隔——通常是元素定位或者其他对元素的操作。

Selenium WebDriver 提供了两类等待事件 (waits): 隐式和显式。

- 显式等待会让 WebDriver 在更深一步的执行前等待一个确定的条件触发。
- 隐式等待则会让 WebDriver 试图定位元素时对 DOM 进行指定次数的轮询。

### 显式等待

显式等待一个确定的条件触发, 然后才进行更深一步的执行。最糟糕的做法是 `time.sleep()`, 指定的条件是等待一个指定的时间段。这里提供了一些便利的方法让编写的代码只等待需要的时间, `WebDriverWait` 结合 `ExpectedCondition (EC)` 是一种实现的方法:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.Firefox()
driver.get("http://www.python.org")
try:
    element = WebDriverWait(driver,10).until(
        EC.presence_of_element_located((By.ID,"about"))
    )
finally:
    driver.quit()
```

这段代码会等待 10 秒, 如果 10 秒内找到元素则立即返回, 否则抛出 `TimeoutException` 异常。`WebDriverWait` 默认每 500 毫秒调用一次 `ExpectedCondition`, 直到它返回成功为止。`ExpectedCondition` 的类型是布尔型的, 成功的返回值就是 `true`, 其他类型的 `ExpectedCondition` 成功的返回值就是 `not null`。

### 预期条件

自动化操作网页时, 有许多频繁使用的通用条件。下面列出的是每一个条件的实现。`Selenium+Python` 提供了许多方便的方法, 因此不需要自己编写 `expected_condition` 的类, 或者创建自己的通用包。`expected_conditions` 模块包含了一系列预定义的条件来和 `WebDriverWait` 联合使用, 下表为具体说明。

条 件	说 明
<code>title_is</code>	判断当前页面的 title 是否精确等于预期



续表

条 件	说 明
<code>title_contains</code>	判断当前页面的 <code>title</code> 是否包含预期字符串
<code>presence_of_element_located</code>	判断某个元素是否被加入 DOM 树中, 并不代表该元素一定可见
<code>visibility_of_element_located</code>	判断某个元素是否可见。可见代表元素非隐藏, 并且元素的宽和高都不等于 0
<code>visibility_of</code>	跟上面的方法做一样的事情, 只是上面的方法要传入 <code>locator</code> , 这个方法直接传定位到 <code>element</code>
<code>presence_of_all_elements_located</code>	判断是否至少有 1 个元素存在于 DOM 树中。举个例子, 如果页面上有 <code>n</code> 个元素的 <code>class</code> 都是 <code>column-md-3</code> , 那么只要有 1 个元素存在, 这个方法就返回 <code>True</code>
<code>text_to_be_present_in_element</code>	判断某个元素中的 <code>text</code> 是否包含了预期的字符串
<code>text_to_be_present_in_element_value</code>	判断某个元素中的 <code>value</code> 属性是否包含了预期的字符串
<code>frame_to_be_available_and_switch_to_it</code>	判断该 <code>frame</code> 是否可以 “switch” 进去, 如果可以, 则返回 <code>True</code> 并且 “switch” 进去, 否则返回 <code>False</code>
<code>invisibility_of_element_located</code>	判断某个元素中是否不存在于 DOM 树或不可见
<code>element_to_be_clickable</code>	判断某个元素中是否可见并且是 <code>enable</code> 的, 这样才叫 <code>clickable</code>
<code>staleness_of</code>	等某个元素从 <code>dom</code> 树中移除, 注意, 这个方法也返回 <code>True</code> 或 <code>False</code>
<code>element_to_be_selected</code>	判断某个元素是否被选中了, 一般用在下拉列表中
<code>element_located_to_be_selected</code>	跟上面的方法作用一样, 只是上面的方法传入定位到的 <code>element</code> , 而这个方法传入 <code>locator</code>
<code>element_selection_state_to_be</code>	判断某个元素的选中状态是否符合预期
<code>element_located_selection_state_to_be</code>	跟上面的方法作用一样, 只是上面的方法传入定位到的 <code>element</code> , 而这个方法传入 <code>locator</code>
<code>alert_is_present</code>	判断页面上是否存在 <code>alert</code>

使用显式等待配合不同的预设置条件就能使我们如同真正浏览网页般操控 `WebDriver`, 举一个例子来说明:

- (1) 打开百度直至网页将其搜索输入框加载成功。
- (2) 在输入框中填写“Scrapy”并单击搜索按钮。
- (3) 等待搜索结果中出现带有“http://www.scrapy.org”的链接。
- (4) 通过链接进入 Scrapy 官网。
- (5) 将 Scrapy 官网页面底部的 GitHub Forks 打印出来。

具体代码如下：

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.PhantomJS()
driver.get("http://www.baidu.com")
try:
    keyword_input = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "#kw"))
    )
    submit = WebDriverWait(driver, 10).until(
        EC.element_to_be_clickable((By.ID, '#su'))
    )
    keyword_input.send_keys('scrapy')
    submit.click()

    scrapy_official_link = WebDriverWait(driver, 10).until(
        EC.text_to_be_present_in_element((By.TagName, 'a'), 'Scrapy')
    )
    scrapy_official_link.click()

    scrapy_forks = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, '#gh-count'))
    )
    print scrapy_forks.text

finally:
    driver.quit()
```



### 隐式 waits

当我们要找一个或者一些不能立即可用的元素时，隐式 waits 会告诉 WebDriver 轮询 DOM 指定的次数，默认设置是 0 次。一旦设定，WebDriver 对象实例的整个生命周期的隐式调用也就设定好了。

```
from selenium import webdriver

driver = webdriver.Firefox()
driver.implicitly_wait(10) # 秒
driver.get("http://somedomain/url_that_delays_loading")
myDynamicElement = driver.find_element_by_id('myDynamicElement')
```

- 显示等待：明确的行为表现，在本地的 Selenium 中运行（选择的编程语言）。可以在任何能想到的条件下工作，返回成功或者超时；可以定义元素的缺失为条件；可以定制重试间隔；可以忽略某些异常。
- 隐式等待：不明确的行为表现，同一个问题在不同的操作系统、不同的浏览器、不同的 Selenium 版本下会有各种不同的表现。在远程 Selenium 上运行（控制浏览器的那部分），只能在寻找元素的函数上工作，返回找到元素或者（在超时以后）没有找到。如果检查元素缺失，那么总是会等待到超时，除了时间什么都不能指定。

#### 4.5.2.4 行为链

```
class selenium.webdriver.common.action_chains.ActionChains(driver)
```

行为链（ActionChains）可以完成简单的交互行为，例如，鼠标移动、鼠标单击事件、键盘输入，以及内容菜单交互。这对于模拟那些复杂的类似于鼠标悬停和拖动行为很有用。

#### 产生用户行为

在 ActionChains 对象上调用行为方法时，这些行为会存储在 ActionChains 对象的一个队列中。调用 perform() 时，这些动作就以它们队列的顺序来触发。

ActionChains 可以使用链式模型：

```
menu = driver.find_element_by_css_selector(".nav")
hidden_submenu = driver.find_element_by_css_selector(".nav #submenu1")

ActionChains(driver).move_to_element(menu).click(hidden_submenu).perform()
```

或者一个个排队，然后执行：

```
menu = driver.find_element_by_css_selector(".nav")
hidden_submenu = driver.find_element_by_css_selector(".nav #submenu1")

actions = ActionChains(driver)
actions.move_to_element(menu)
actions.click(hidden_submenu)
action.perform()
```

不管怎样，这些动作总是一个接一个地按它们被调用的顺序执行。

行为 API 参考如下表所示。

方 法	说 明	参 数 说 明
click(on_element=None)	单击一个元素	on_element: 要单击的元素，如果是 None，则单击鼠标当前的位置
click_and_hold(on_element=None)	鼠标左键单击一个元素并且保持	on_element: 同 click() 类似
double_click(on_element=None)	双击一个元素	on_element: 同 click() 类似
drag_and_drop(source, target)	鼠标左键单击 source 元素，然后移动到 target 元素释放鼠标按钮	source: 鼠标单击的元素 target: 鼠标松开的元素
drag_and_drop_by_offset(source, xoffset, yoffset)	拖动目标元素到指定的偏移点释放	source: 单击的参数 xoffset: X 偏移量 yoffset: Y 偏移量
key_down(value, element=None)	只按下键盘，不释放。我们应该只对那些功能键使用 (Control、Alt、Shift) value: 要发送的键，值在 Keys 类中有定义 element: 发送的目标元素，如果是 None，value 会发到当前聚焦的元素上。例如：我们要按下 ctrl+c: ActionChains(driver).key_down(Keys.CONTROL).send_keys('c').key_up(Keys.CONTROL).perform()	



续表

方 法	说 明	参 数 说 明
key_up(value, element=None)	释放键	参考 key_down 的解释
move_by_offset(xoffset, yoffset)	将当前鼠标的位置进行移动	xoffset: 要移动的 X 偏移量, 可以是正, 也可以是负 yoffset: 要移动的 Y 偏移量, 可以是正, 也可以是负
move_to_element(to_element)	把鼠标移到一个元素的中间	to_element: 目标元素
move_to_element_with_offset(to_element, xoffset, yoffset)	鼠标移动到元素的指定位置, 偏移量以元素的左上角为基准	to_element: 目标元素 xoffset: 要移动的 X 偏移量 yoffset: 要移动的 Y 偏移量
perform()	执行所有存储的动作	
Release(on_element=None)	释放一个元素上的鼠标按键	on_element: 如果为 None, 则在当前鼠标位置上释放
send_keys(*keys_to_send)	向当前的焦点元素发送键	keys_to_send: 要发送的键, 修饰键可以到 Keys 类中找到
send_keys_to_element(element, *keys_to_send)	向指定的元素发送键	

4.5.2.5 Cookies

在 Selenium 中处理 Cookies 是常用的一项功能, 尤其是对于那些用 Cookies 记录访客身份或者以此作为访问痕迹跟踪的网站, 在客户端维护 Cookie 就显得尤为必要了。

以下是 Selenium 获取 Cookies 的方法, 以访问某电商网站为例:

```
from selenium import webdriver

driver = webdriver.PhantomJS()
driver.get("https://www.某电商网站网址.com")
driver.implicitly_wait(1)

print driver.get_cookies()
```

可以调用 delete\_cookie()、add\_cookie() 和 delete\_all\_cookies() 方法来处理 Cookie。另外, 还可以保存 Cookie 以备其他网络爬虫使用。下面的例子演示了如何把这些函数

组合在一起:

```
from selenium import webdriver

driver = webdriver.PhantomJS()
driver.get("https://www.某电商网站网址.com")
driver.implicitly_wait(1)

print driver.get_cookies()

savedCookies = driver.get_cookies()

driver2 = webdriver.PhantomJS()
driver2.get("https://www.某电商网站网址.com")
driver2.delete_all_cookies()

[driver.add_cookie(cookie) for cookie in savedCookies]
driver2.get("https://www.某电商网站网址.com")
driver2.implicitly_wait(1)
print driver2.get_cookies()
```

在这个例子中, 第一个 WebDriver 获得了一个网站, 打印 Cookie 并把它们保存到变量 savedCookies 中。第二个 WebDriver 加载同一个网站 (必须首先加载网站, 这样 Selenium 才能知道 Cookie 属于哪个网站, 即使加载网站的行为对我们没任何用处), 删除所有 Cookie, 然后替换成第一个 WebDriver 得到的 Cookie。当再次加载这个页面时, 两组 Cookie 的时间戳、源代码和其他信息应该完全一致。从 Google Analytics 的角度看, 第二个 WebDriver 现在和第一个 WebDriver 完全一样。

在高阶虫术关于反跟踪的一节中, 我们会继续深入 Cookies 数据内部进行一番探讨, 在此阶段只需要知道如何在 Selenium 中对 Cookies 进行相关的处理就够了。

#### 4.5.2.6 将爬虫接入Selenium

前面的章节主要讲述 Selenium 的基本组成部分及其应用的方法。然而一项技术没有被完全实践应用之前只能停留于对概念的认知, 本节会把 Selenium 接入本章开篇时那个尚未完成的爬虫示例中。

Scrapy 之所以强大, 是因为它各个部分都可以被独立扩展, 这种松散的结构使得每个部分的开发都极为灵活。我们要使其支持 JavaScript 网页解释, 最绝妙的方法不是去修改蜘蛛的代



码, 而是向这个项目“插入”一个功能部件, 增强其自身的能力, 无须改变原有代码的一丝一毫。

要做到向项目增加一项功能而无须更改蜘蛛的代码, 就需要编写一个下载器中间件了, 这个中间件只要正常地导航到产品页面并对 JavaScript 做出正确的解释, 然后将 JavaScript 渲染后的网页内容传递给蜘蛛即可。具体思路: 在 Middleware 的 `process_request()` 方法中对每个抓取请求进行处理, 启动浏览器并进行页面渲染, 再将渲染后的结果构造一个 `HtmlResponse` 返回即可。

首先我们在 `__init__()` 中对一些对象进行初始化, 包括 `PhantomJS`、`WebDriverWait` 等对象, 同时设置页面大小和页面加载超时时间, 随后在 `process_request()` 方法中先通过 `request` 的 `meta` 属性获取当前需要爬取的页码, 然后调用 `PhantomJS` 对象的 `get()` 方法访问 `request` 的对应的 URL, 相当于从 `Request` 对象中获取了请求链接后再用 `PhantomJS` 去加载, 而不再使用 `Scrapy` 中的 `Downloader`。

将 `SELENIUM_TIMEOUT`(`Selenium` 连接超时)和 `PHANTOMJS_SERVICE_ARGS`(`PhantomJS` 的服务配置变量)从配置文件中读入, 可以使我们的中间件变得可以配置, 使用起来更加灵活。

```
@classmethod
def from_crawler(cls, crawler):
    return cls(timeout=crawler.settings.get('SELENIUM_TIMEOUT'),
               service_args=crawler.settings.get('PHANTOMJS_SERVICE_ARGS'))

def __init__(self, timeout=None, service_args=[]):
    self.logger = getLogger(__name__) # 打开日志
    self.timeout = timeout
    # 设置 PhantomJS 的运行参数
    self.browser = webdriver.PhantomJS(service_args=service_args)
    self.browser.set_window_size(1400, 700) # 设置浏览窗口
    # 设置浏览器加载网页的超时时间
    self.browser.set_page_load_timeout(self.timeout)
    self.wait = WebDriverWait(self.browser, self.timeout)
```

另外我们要在析构函数内手动地释放浏览器实例:

```
def __del__(self):
    self.browser.close() # 析构时关闭浏览器实例
```

然后, 在 `process_request` 函数内写入处理逻辑:

```

# 在浏览器打开网页
self.browser.get(request.url)

# 等待页面的宝贝全部加载完成
self.wait.until(EC.presence_of_element_located((By.CSS_SELECTOR,
'.m-itemlist .items .item'))))

# 用 Phantom 解释后的网页结果构造新的 HtmlResponse
return HtmlResponse(url=request.url,
                    body=self.browser.page_source,
                    request=request,
                    encoding='utf-8',
                    status=200)

```

等待页面加载完成之后,调用 PhantomJS 的 `page_source` 属性即可获取当前页面的源代码,用它来直接构造了一个 `HtmlResponse` 对象并返回。

完整代码如下所示。

```

from selenium import webdriver
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from scrapy.http import HtmlResponse
from logging import getLogger

class SeleniumMiddleware():

    @classmethod
    def from_crawler(cls, crawler):
        return cls(timeout=crawler.settings.get('SELENIUM_TIMEOUT'),
                  service_args=crawler.settings.get('PHANTOMJS_SERVICE_ARGS'))

    def __init__(self, timeout=None, service_args=[]):
        self.logger = getLogger(__name__) # 打开日志
        self.timeout = timeout
        self.browser = webdriver.PhantomJS(service_args=service_args)

```



```

self.browser.set_window_size(1400, 700) # 设置浏览器窗口
# 设置浏览器加载网页的超时时间
self.browser.set_page_load_timeout(self.timeout)
self.wait = WebDriverWait(self.browser, self.timeout)

def __del__(self):
    self.browser.close() # 释构时关闭浏览器实例

def process_request(self, request, spider):
    """
    用 PhantomJS 抓取页面
    :param request: request 对象
    :param spider: Spider 对象
    :return: HtmlResponse
    """
    self.logger.debug(u'启动 PhantomJS...')
    # page = request.meta.get('sn', 1)

    try:
        self.browser.get(request.url)

        # 等待页面的宝贝全部加载完成
        self.wait.until(EC.presence_of_element_located((By.CSS_SELECTOR, '.m-itemlist .items .item'))))

        return HtmlResponse(url=request.url,
                             body=self.browser.page_source,
                             request=request,
                             encoding='utf-8',
                             status=200)

    except TimeoutException:
        # 超时抛出异常
        return HtmlResponse(url=request.url, status=500, request=request)

```

这里可能有人会纳闷了, 为什么通过实现 Downloader Middleware 就可以了呢? 之前的 request 对象怎么办? Scrapy 不再处理了吗? Response 返回后又传递给了谁来进行处理?

是的, `request` 对象到这里就不会再处理了, 也不会再像以前一样交给 `Downloader` 下载了, `response` 会直接传给 `Spider` 进行解析。

这究竟是因为什么呢? 这时我们需要回顾一下 `Downloader Middleware` 的 `process_request()` 方法的处理逻辑, 在前面我们也提到过, 内容如下:

当 `process_request()` 方法返回 `response` 对象时, 更低优先级的 `Downloader Middleware` 的 `process_request()` 和 `process_exception()` 方法就不会被继续调用了, 转而依次开始执行每个 `Downloader Middleware` 的 `process_response()` 方法, 调用完毕之后直接将 `response` 对象发送给 `Spider` 来处理。

在这里直接返回了一个 `HtmlResponse` 对象, 它是 `response` 的子类, 同样满足此条件, 返回之后便会按顺序调用每个 `Downloader Middleware` 的 `process_response()` 方法。而在 `process_response()` 中, 我们没有对其做特殊处理, 接着它就会被发送给 `Spider`, 传给 `request` 的回调函数进行解析。

到现在我们应该就能了解 `Downloader Middleware` 实现 `Selenium` 对接的原理了。

在 `settings.py` 中开启它的调用:

```
DOWNLOADER_MIDDLEWARES = {
    'taobao.middlewares.SeleniumMiddleware': 543,
}
```

## 小结

在 `Selenium` 的实践中应用的内容其实并不多, 而前几节的内容是给出一个全面性的应用概念, 当遇到相对复杂的交互场景时, 它们自然就会发挥其应有的作用。然而, `Selenium` 更多地被应用在白盒测试场景, 在爬虫系统中是不应该出现太过复杂的交互的。因为交互只是为了得到最终的 URL 而已, 所以很多地方可以跳到交互的结果上直接计算出其 URL。也就是说, `Selenium` 的运用代码应该越少越好, 过多的交互代码有可能预示着代码思路出现了问题, 这是必须引起警惕的。

## 4.5.3 Scrapy与Splash

### Splash

`Splash` 是一个 JavaScript 渲染服务。它基于 `Twisted` 和 `QT5`, 采用 `Python 3` 实现的带有 HTTP API 的轻量级网页浏览器。由于 `Python` 本身是没有多线程概念的, 而 `Twisted` 的 `reactor` 可以使 `QT` 主循环中的 `WebKit` 并行化处理发挥得淋漓尽致。



- 并行处理多个网页;
- 获得 HTML 处理结果或者进行屏幕截取;
- 采用 Adblock Plus 的规则消除广告相关的图片以加速网页渲染的速度;
- 在页面上下文内执行定义的 JavaScript 脚本;
- 通过 Lua 来执行脚本;
- 在 Splash-Jupyter Notebooks 内用 Lua 脚本开发 Splash 应用;
- 获取渲染 HAR 格式内容的详细信息。

scrapy-splash 与 Selenium+WebDriver 相比, 优势有以下几点:

- Splash 作为 JS 渲染服务, 是基于 Twisted 和 QT 开发的轻量浏览器引擎, 并且提供直接的 HTTP API。快速、轻量的特点使其容易进行分布式开发。
- Splash 和 Scrapy 融合, 两种互相兼容彼此的特点, 抓取效率较好。
- Splash 的线程模型是非阻塞式的, 与 Selenium 的阻塞式处理相比有更明显的性能提升, 速度上更优于 Selenium。

#### ➤ 安装 Splash

安装 Splash 最简单办法就是直接设置一个 Docker 虚拟机, 具体做法如下:

```
$ docker pull scrapinghub/splash
$ docker run -p 5023:5023 -p 8050:8050 -p 8051:8051 scrapinghub/splash
```

这样 Splash 服务就运行于 localhost:8050 端口了。

#### ➤ Splash HTTP API

Splash 是通过 HTTP API 提供服务的, 以下是 Splash 服务终结点 (EndPoints)。

- render.json——返回一个 JSON 编码的字典, 其中包含 JavaScript 渲染后的网页信息。基于传递的参数, 它可以包含 HTML、PNG 和其他信息。
- render.html——将 JavaScript 渲染结果以 HTML 格式返回。
- render.png——将 JavaScript 渲染结果以 PNG 图片格式返回。
- render.jpeg——将 JavaScript 渲染结果以 JPEG 图片格式返回。
- render.har——以 HAR 格式返回有关 Splash 与网站交互的信息, 包括有关请求、收到的响应、计时、标题等的信息。

HAR (HTTP Archive) 是一个用来储存 HTTP 请求/响应信息的通用文件格式, 基于 JSON。这个格式的出现可以使 HTTP 监测工具以一种通用的格式导出所收集的数据, 这些数据可以被其他支持 HAR 的 HTTP 分析工具(包括 Firebug、httpwatch、Fiddler 等)所使用, 用来分析网站的性能瓶颈。目前 HAR 规范最新版本为 HAR 1.2。HAR 文件必须是 UTF-8 编码, 有无 BOM 无所谓。一个 HAR 文件就是一个 JSON 对象, 如下:

```
{
  "log": {
    "version" : "1.2",
    "creator" : {},
    "browser" : {},
    "pages": [],
    "entries": [],
    "comment": ""
  }
}
```

- version [string]——版本号, 默认为 1.1。
- creator [object]——创建 HAR 文件的程序名称和版本信息。
- browser [object, 可选]——浏览器的名称和版本信息。
- pages [array, 可选]——页面列表, 如果应用不支持按照 page 分组, 则可以省去此字段。
- entries [array]——所有 HTTP 请求的列表。
- comment [string, 可选] (new in 1.2) ——注释。

**注:** 每个页面对应一个对象, 每个 HTTP 请求对应一个对象。如果 HTTP 的监测分析工具不能把请求按照 page 分组, 则为空。

我们可以用最简单的办法来测试一下 Splash 是否已正常运作了, 在浏览器内直接访问 Splash 的 API 的终结点, 打开浏览器并键入:

```
http://localhost:8050/render.html?url=http://www.taobao.com
```

如下图所示, 代表访问淘宝成功。





然后换成 `render.json` 的终结点，在命令行下运行：

```
$ curl http://localhost:8050/render.json?url=http://www.taobao.com
```

会得到这样的输出结果：

```
{
  "url": "https://www.taobao.com/",
  "title": "\u6dd8\u5b9d\u7f51 - \u6d88\u60a0\u7f51",
  "geometry": [0, 0, 1024, 768],
  "requestedUrl": "http://www.taobao.com/"
}
```

如果换成 `render.jpeg` 或者 `render.png` 的终结点，则会得两个淘宝网网页的截屏图片，是否很有趣？

最后换成 `render.har` 的终结点，会看到一份更详细的 JSON 结果：

```
{
  "log": {
    "pages": [{
      "id": "1",
      "startedDateTime": "2018-01-01T05:20:44.901401Z",
```

```

        "title": "\u6dd8\u5b9d\u7f51 - \u6dd8\u5f01\u6211\u559c\u6b22",
        "pageTimings": {"onContentLoaded": 213, "_onPrepareStart": 348,
        "_onStarted": 0, "onLoad": 348}
    }, {"version": "1.2", "browser": {
        "version": "602.1", "comment": "PyQt 5.9, Qt 5.9.1", "name":
        "QWebKit"
    }
    ,
    "creator": {
        "version": "3.0", "name": "Splash"
    }
    ,
    "entries": [{
        "request": {
            "cookies": [],
            "url": "http://www.taobao.com/",
            "httpVersion": "HTTP/1.1",
            "method": "GET",
            "bodySize": -1,
            "headers": [{
                "name": "User-Agent",
                "value": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/
602.1 (KHTML, like Gecko) splash Version/9.0 Safari/602.1"
            }, {"name": "Accept", "value": "text/html,application/xhtml+
xml,application/xml;q=0.9,*/*;q=0.8"}],
            "headersSize": 188,
            "queryString": []
        },
        "response": {
            "content": {"mimeType": "text/html", "size": 258},
            "bodySize": 258,
            "ok": true,
            "status": 302,
            "redirectURL": "https://www.taobao.com/",
            "cookies": [{
                "expires": "2019-01-01T05:20:45Z",
                "path": "/",
                "domain": ".taobao.com",

```



```

        "secure": false,
        "name": "thw",
        "value": "cn",
        "httpOnly": false
    }],
    "headersSize": 290,
    "statusText": "Found",
    "headers": [{ "name": "Server", "value": "Tengine", {
        "name": "Date",
        "value": "Mon, 01 Jan 2018 05:20:45 GMT"
    }, { "name": "Content-Type", "value": "text/html", {
        "name": "Content-Length",
        "value": "258"
    }, { "name": "Connection", "value": "keep-alive", {
        "name": "Location",
        "value": "https://www.taobao.com/"
    }, {
        "name": "Set-Cookie",
        "value": "thw=cn; Path=/; Domain=.taobao.com; Expires=Tue,
01-Jan-19 05:20:45 GMT;"
    }, { "name": "Strict-Transport-Security", "value": "max-age=
31536000"}]],
    "httpVersion": "HTTP/1.1",
    "url": "http://www.taobao.com/"
},
"cache": {},
"_splash_processing_state": "finished",
"pageref": "1",
"startedDateTime": "2018-01-01T05:20:44.902180Z",
"time": 52,
"timings": { "send": 1, "wait": 51, "receive": 0, "blocked": -1,
"ssl": -1, "dns": -1, "connect": -1}
}, {
    "request": {
        "cookies": [{
            "path": "",
            "domain": "",
            "secure": false,

```

```

        "name": "thw",
        "value": "cn",
        "httpOnly": false
    }],
    "url": "https://www.taobao.com/",
    "httpVersion": "HTTP/1.1",
    "method": "GET",
    "bodySize": -1,
    "headers": [{
        "name": "User-Agent",
        "value": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/
602.1 (KHTML, like Gecko) splash Version/9.0 Safari/602.1"
    }, {
        "name": "Accept",
        "value": "text/html,application/xhtml+xml,application/xml;
q=0.9,*/*;q=0.8"
    }, {"name": "Cookie", "value": "thw=cn"}],
    "headersSize": 204,
    "queryString": []
},
"response": {
    "content": {"mimeType": "text/html; charset=utf-8", "size": 0},
    "bodySize": 128031,
    "ok": true,
    "status": 200,
    "redirectURL": "",
    "cookies": [],
    "headersSize": 664,
    "statusText": "OK",
    "headers": [{"name": "Server", "value": "Tengine"}, {
        "name": "Date",
        "value": "Mon, 01 Jan 2018 05:20:45 GMT"
    }, {"name": "Content-Type", "value": "text/html; charset=
utf-8"}, {
        "name": "Transfer-Encoding",
        "value": "chunked"
    }, {"name": "Connection", "value": "keep-alive"}, {
        "name": "Vary",

```



```

        "value": "Accept-Encoding, Ali-Detector-Type"
    }, {"name": "Cache-Control", "value": "max-age=60,
s-maxage=90"}, {
        "name": "X-Snapshot-Age",
        "value": "1"
    }, {"name": "ETag", "value": "W/\\"2934-1602480fa54\\\"", {
        "name": "Via",
        "value": "cache28.12cn41[12,304-0,H], cache3.12cn41[13,0],
cache1.cn200[0,200-0,H], cache2.cn200[1,0]"
    }, {"name": "X-Swift-Error", "value": "forward peer reset"}, {
        "name": "Age",
        "value": "60"
    }, {"name": "X-Cache", "value": "HIT TCP_MEM_HIT dirn:-2:-2
mlen:-1"}, {
        "name": "X-Swift-SaveTime",
        "value": "Mon, 01 Jan 2018 05:19:45 GMT"
    }, {"name": "X-Swift-CacheTime", "value": "90"}, {
        "name": "Timing-Allow-Origin",
        "value": "*"
    }, {"name": "EagleId", "value": "3d83271815147840452945983e"}, {
        "name": "Strict-Transport-Security",
        "value": "max-age=31536000"
    }, {"name": "Content-Encoding", "value": "gzip"}],
    "httpVersion": "HTTP/1.1",
    "url": "https://www.taobao.com/"
},
"cache": {},
"_splash_processing_state": "finished",
"pageref": "1",
"startedDateTime": "2018-01-01T05:20:44.953594Z",
"time": 181,
"timings": {"send": 1, "wait": 125, "receive": 55, "blocked": -1,
"ssl": -1, "dns": -1, "connect": -1}
}, {
    "request": {
        "cookies": [],
        "url": "https://log.mmstat.com/eg.js",
        "httpVersion": "HTTP/1.1",

```

```

"method": "GET",
"bodySize": -1,
"headers": [{"name": "Referer", "value": "https://www.
taobao.com/"}], {
    "name": "If-None-Match",
    "value": "\"tK3QEu4gzmACATo/k06p+Fv/\""
}, {
    "name": "User-Agent",
    "value": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/
602.1 (KHTML, like Gecko) splash Version/9.0 Safari/602.1"
}, {"name": "Accept", "value": "*/*"}],
"headersSize": 205,
"queryString": []
},
"response": {
    "content": {"mimeType": "application/javascript", "size": 91},
    "bodySize": 91,
    "ok": true,
    "status": 200,
    "redirectURL": "",
    "cookies": [{
        "expires": "2027-12-30T05:20:45Z",
        "path": "/",
        "domain": ".mmstat.com",
        "secure": false,
        "name": "cna",
        "value": "tK3QEu4gzmACATo/k06p+Fv/",
        "httpOnly": false
    }],
    "headersSize": 352,
    "statusText": "OK",
    "headers": [{"name": "Date", "value": "Mon, 01 Jan 2018 05:20:45
GMT"}], {
    "name": "Content-Type",
    "value": "application/javascript"
}, {"name": "Content-Length", "value": "91"}, {
    "name": "Connection",
    "value": "keep-alive"

```



```

    }, {"name": "ETag", "value": "\"tK3QEu4gzMACATo/k06p+Fv/\"", {
      "name": "stag",
      "value": "0"
    }, {
      "name": "Set-Cookie",
      "value": "cna=tK3QEu4gzMACATo/k06p+Fv/; expires=Thu,
30-Dec-27 05:20:45 GMT; path=/; domain=.mmstat.com"
    }, {"name": "Expires", "value": "Thu, 01 Jan 1970 00:00:01 GMT"}, {
      "name": "Cache-Control",
      "value": "no-cache"
    }, {"name": "Pragma", "value": "no-cache"}],
    "httpVersion": "HTTP/1.1",
    "url": "https://log.mmstat.com/eg.js"
  },
  "cache": {},
  "_splash_processing_state": "finished",
  "pageref": "1",
  "startedDateTime": "2018-01-01T05:20:45.121500Z",
  "time": 128,
  "timings": {"send": 0, "wait": 128, "receive": 0, "blocked": -1,
"ssl": -1, "dns": -1, "connect": -1}
  }}
}
}

```

这份 HAR 格式的 JSON 文件会给爬网分析带来巨大的帮助。

#### 4.5.3.1 scrapy-splash

Scrapy 只是提供了一个 Web 渲染服务, 如果在 Scrapy 中直接使用, 则效果与 Selenium 差不多, 但有了 scrapy-splash 开发工具包就大不一样了, 它是将 Scrapy 与 Splash 完美结合一体的工具包, scrapy-splash 搭载了一系列的中间件和针对 Scrapy 开发的扩展, 以达到开箱即用的效果。

源码地址: <https://github.com/scrapy-plugins/scrapy-splash>。

#### 安装

按照以下指令通过 pip 安装 scrapy-splash:

```
$ pip install scrapy-splash
```

scrapy-splash 直接使用 Splash 的 HTTP API, 所以在使用 scrapy-splash 之前需要一个 Splash 实例。按前文所述, 我们只要启动一个 Splash 的 Docker 实例就可以了:

```
$ docker run -p 8050:8050 scrapinghub/splash
```

## 配置

首先要在 Scrapy 工程的配置文件 settings.py 中添加 Splash 服务器的访问地址:

```
SPLASH_URL = 'http://localhost:8050'
```

如果 Splash 运行在云端, 那么上述地址就应该是该服务器的外部访问地址。

然后添加 Splash 中间件, 在 settings.py 中使用 DOWNLOADER\_MIDDLEWARES 配置项指定, 要加入 SplashCookiesMiddleware 和 SplashMiddleware 两个中间件, 由于这个中间件要求在 HttpCompressionMiddleware 中间件之前运行, 因此需要修改 HttpCompressionMiddleware 的默认优先级, 具体配置如下:

```
DOWNLOADER_MIDDLEWARES = {
    'scrapy_splash.SplashCookiesMiddleware': 723,
    'scrapy_splash.SplashMiddleware': 725,
    'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware':
810,
}
```

默认情况下, HttpProxyMiddleware 的优先级是 750, 要把它放在 Splash 中间件后面。

最后, scrapy-splash 内置了针对 Splash 专用的重过滤器, 按以下方式配置可以启用 Splash 的去重功能:

```
DUPEFILTER_CLASS = 'scrapy_splash.SplashAwareDupeFilter'
```

另外, 如果想使用 Splash 的 HTTP 缓存, 则还需指定一个自定义的缓存后台存储介质, scrapy-splash 内置了一个 scrapy.contrib.httptcache.FilesystemCacheStorage 的子类, 可以在 settings.py 文件中启用它, 代码如下所示。

```
HTTPCACHE_ENABLED = True
```



```
HTTPCACHE_STORAGE = 'scrapy_splash.SplashAwareFSCacheStorage'
```

如果要在 Splash 中使用其他缓存存储,则需要继承这个类并将所有 `scrapy.util.request.request_fingerprint` 调用替换成 `scrapy_splash.splash_request_fingerprint`。

## 其他配置项

scrapy-splash 还提供了一些其他的可选的配置项:

- `SPLASH_COOKIES_DEBUG`——默认为 `False`。当设置为 `True` 时,可启用 `SplashCookiesMiddleware` 中间件的 Cookie 调试功能。这个选项有点类似于 Scrapy 内置的 Cookie 中间件的 `COOKIES_DEBUG` 选项的功能。它将所有发出或者收到的 Cookies 记录到日志中。
- `SPLASH_LOG_400`——默认为 `True`。记录所有 HTTP 的 400 错误,这样做非常重要,因为我们可以从日志中分析得出 Splash 执行脚本时可能出现的问题。
- `SPLASH_SLOT_POLICY`——默认为 `scrapy_splash.SlotPolicy.PER_DOMAIN`。用于统一指定 Splash 将执行何种并发策略,并作为下文提及的 `SplashRequest` 对象的 `slot_policy` 参数的默认值。

## 编写蜘蛛

现在, Splash 的应用中间件已配置完成,可以在 Scrapy 中使用 `SplashRequest` 对象取代原生的 `request` 发起网页请求。

例如,同之前在 `start_requests` 函数中生成 `request` 一样,只要直接用 `SplashRequest` 将其取代,就可以将请求交由 Splash 服务器进行处理并生成正确的渲染结果返回。

首先在项目中创建一个蜘蛛,指令如下所示。

```
$ scrapy genspider taobao taobao.com
```

然后打开项目目录下 `spiders` 目录中的 `taobao.py`,添加以下代码:

```
import scrapy
from scrapy_splash import SplashRequest

class TaobaoSpider(scrapy.Spider):
    name = 'taobao'
    allowed_domains = ['taobao.com']
```

```

start_urls = ['http://taobao.com/']

def start_requests(self):
    for url in self.start_urls:
        yield SplashRequest(url, self.parse,
                             endpoint='render.html',
                             args={'wait': 0.5},
                             )

def parse(self, response):
    pass

```

SplashRequest 中的 args 属性字典会将所有参数一并发送到 Splash 服务器上。默认情况下, endpoint 是指向 render.json 文件的, 但这里在实例化 SplashRequest 时直接将默认参数进行覆盖, 让 Splash 将结果渲染到 render.html 文件中以提供一个 HTML 的响应结果。

#### 4.5.3.2 Splash的请求对象

最简单的渲染请求的方式是使用 scrapy\_splash.SplashRequest, 通常应该选择使用 yield SplashRequest:

```

yield SplashRequest(url, self.parse_result,
                    args={
                        # 此字典为向 Splash HTTP API 传选的可选参数;
                        'wait': 0.5,
                        # 'url' 每个请求预先填充的目标请求地址
                        # 'http_method' 用于设置请求方法, 如 POST、PUT
                        # 'body' 设置用于 POST 方法时向服务端发起的请求正文
                    },
                    endpoint='render.json',      # 可选; 默认为 render.html
                    splash_url='<url>',          # 可选; 用于覆盖配置文件中的 SPLASH_URL 选项
                    slot_policy=scrapy_splash.SlotPolicy.PER_DOMAIN, # 可选
                    )

```

另外, 还可以在普通的 Scrapy 请求中传递 Splash 请求 meta 关键字以实现同样的效果。

```

yield scrapy.Request(url, self.parse_result, meta={
    'splash': {

```



```

        'args': {
            'html': 1,
            'png': 1
        },
        'endpoint': 'render.json',
        'splash_url': '<url>',
        'slot_policy': scrapy_splash.SlotPolicy.PER_DOMAIN,
        'splash_headers': {},
        'dont_process_response': True,
        'dont_send_headers': True,
        'magic_response': False
    }
})

```

### 执行JavaScript脚本

可以在 `SplashRequest` 的构造函数中将要执行的 JavaScript 代码字符串传给 `js_source` 参数。请求被正常发出并返回后，当页面的加载完成且未被渲染之前，传入 `js_source` 参数的 JavaScript 脚本会被执行。这一特性允许使用 JavaScript 在页面渲染之前修改页面内的 DOM 对象，代码如下所示。

```

yield SplashRequest(
    'http://example.com',
    endpoint='render.html',
    args={'js_source': 'document.title="自定义标题";'},
)

```

### Splash meta说明

本质上 `SplashRequest` 是用来填充 `request.meta['splash']` 数据的工具类，`SplashRequest` 会生成一个 `request` 并将自身的属性自动填充到以下 `meta['splash']` 属性中：

- `meta['splash']['args']`——包含发往 Splash 的参数。
- `meta['splash']['endpoint']`——指定 Splash 所使用的 endpoint，默认是 `render.html`。
- `meta['splash']['splash_url']`——覆盖 `settings.py` 文件中配置的 Splash URL。
- `meta['splash']['splash_headers']`——运行增加或修改发往 Splash 服务器的 HTTP 头部信息，注意不是修改发往远程 Web 站点的 HTTP 头部信息。
- `meta['splash']['dont_send_headers']`——如果不想传递 headers 给 Splash，则将

它设置成 True。

- `meta['splash']['slot_policy']`——指定如何让 Splash 维护请求的并发性。
- `meta['splash']['dont_process_response']`——当设置为 True 后，`SplashMiddleware` 不会修改默认的 `scrapy.Response` 请求。默认会返回 `SplashResponse` 子类响应，比如 `SplashTextResponse`。
- `meta['splash']['magic_response']`——默认为 True，Splash 会自动设置 `response` 的一些属性，比如 `response.headers`、`response.body` 等。

如果想通过 Splash 来提交 Form 请求，则可以使用 `scrapy_splash.SplashFormRequest`，它和 `SplashRequest` 的使用方式是一样的。

### 4.5.3.3 Splash的响应对象

对于不同的 Splash 请求，`scrapy-splash` 返回不同的 `response` 子类。

- `SplashResponse`——二进制响应，比如对 `/render.png` 的响应。
- `SplashTextResponse`——文本响应，比如对 `/render.html` 的响应。
- `SplashJsonResponse`——JSON 响应，比如对 `/render.json` 或使用 Lua 脚本的 `/execute` 的响应。

如果只想使用标准的 `Response` 对象，则设置 `meta['splash']['dont_process_response'] = True`。

所有 `response` 会把 `response.url` 设置成原始请求 URL（也就是要渲染的页面 URL），而不是 Splash endpoint 的 URL 地址。实际地址是通过 `response.real_url` 得到的。

`SplashJsonResponse` 还提供了一些特殊的功能：

- `response.data` 属性包含从 JSON 解码的响应数据。
- 如果配置了 Splash 会话处理，则可以使用 `response.cookiejar` 访问当前的 Cookie，它是一个 `CookieJar` 实例。
- 如果请求中启用了 Scrapy-Splash “魔术响应”（默认），则会从原始响应正文中自动设置以下几个响应属性（标题、主体、URL、状态码）：
  - `response.headers` 填充 `headers` 中的键；
  - `response.url` 设置成原始请求 URL；
  - `response.body` 响应的正文内容；
  - `response.status` 返回响应的状态码。



### 4.5.3.4 Splash脚本

Splash 作为一个无头浏览器服务,是否能像 PhantomJS 那样以编程方式来控制浏览器的行为,使得客户端的访问行为看起来更像人类呢? Splash 在 Python 世界是如此的流行,那答案是显而易见的,只是 Splash 的实现方式相对于 PhantomJS 来说有点奇怪。

Selenium+PhantomJS 实际上是通过在 JavaScript 脚本中调用 PhantomJS 的对象模型以编码方式来控制浏览器行为的,在 Python 中也能使用,是因为 Selenium 实现了一个 Python 编程接口的包装。Splash 其实也能通过脚本实现以编程方式控制浏览器行为,但它最不友好的地方是 Splash 脚本是用 Lua 写的,那就意味着除了得懂得 Python 和 JavaScript,还得掌握 Lua。虽然 Lua 是 C++ 的一个简化版本的脚本式语言,而且在游戏开发、嵌入式开发领域中也广泛应用,但它的引入必然会让 Splash 的学习曲线变得陡峭。

Splash 与 Scrapy 结合后优越的性能是值得我们花点时间去学习 Lua 的一个理由。

Splash 在其服务端支持 Lua 脚本,这是执行 JavaScript 的首选方法,因为可以在 Lua 中调用 Splash 预加载库中的对象模型以实现浏览器进行深度控制的效果。

首先来看一个发送到 Splash 端执行的 Lua 脚本,代码如下所示。

```
function main(splash)
    assert(splash:go(splash.args.url)) -- 将浏览器导航至指定 URL
    splash:wait(0.5)                    -- 等待
    local title = splash:evaljs("document.title") -- 在浏览器执行 JavaScript
    脚本将返回当前网页的标题
    return {title=title} -- 将计算结果转换为 JSON 并返回
end
```

每个 Splash 脚本中必须有一个 main(splash)函数作为脚本启动的入口函数,传入 main 函数的 splash 参数实例可以控制浏览器的行为。

上述代码段是一个 Lua 脚本,那是怎么传送到 Splash 执行的呢? 在 Python 中执行这段代码感觉很奇怪,在同一个 Python 代码中我们得用 3 种语言来编写代码? 其实细想也不稀奇, Lua 是一种为简化 C++ 而生的脚本,而控制 Splash 的原生脚本语言就是 Lua,因此我们就只能在 Python 中将上述代码保存为一个字符串并作为参数传递至 Splash 中执行。具体做法如下所示。

```
lua_script = """
function main(splash)
    assert(splash:go(splash.args.url))
    splash:wait(0.5)
```

```

    local title = splash:evaljs("document.title")
    return {title=title}
end
"""

```

```

class MySplashSpider(Spider):

```

```

    def start_requests(self):
        yield SplashRequest(url,
                             callback=self.parse,
                             endpoint='execute',
                             args={
                                 'lua_source': lua_script,
                                 'page': page,
                                 'wait': 7
                             })

```

```

    def parse(self, response):
        # ... 省略

```

入口函数 (main()) 可以返回一个 Lua 表格, 这个表格被渲染成 JSON 对象。我们使用 splash:go 函数来告诉 Splash 访问 URL。splash:evaljs 函数可以在页面上下文中执行 JavaScript。如果不需要返回结果, 则可以使用 splash:runjs 代替 splash:evaljs。

通常情况下, 可能需要在显示页面之前单击按钮。我们可以使用 splash:mouse\_click 函数来实现:

```

function main(splash)
    assert(splash:go(splash.args.url))
    local get_dimensions = splash:jsfunc([[
        function () {
            var rect = document.getElementById('button').getClientRects()[0];
            return {"x": rect.left, "y": rect.top}
        }
    ]])
    splash:set_viewport_full()
    splash:wait(0.1)
    local dimensions = get_dimensions()

```



```

    splash:mouse_click(dimensions.x, dimensions.y)
    -- Wait split second to allow event to propagate.
    splash:wait(0.1)
    return splash:html()
end

```

这里使用 `splash:jsfunc` 来定义一个 JavaScript 函数，它将返回元素的坐标，然后使用 `splash:set_viewport_full` 函数确保元素是可见的，并单击元素。Splash 最后返回呈现的 HTML (`splash:html`)。

附：Splash Lua脚本对象如下表所示

方 法	说 明
<code>splash:set_result_status_code</code>	允许改变 HTTP 结果的状态码
<code>splash:set_result_content_type</code>	允许更改返回给客户端的 Content-Type
<code>splash:set_result_header allows</code>	将自定义 HTTP 标头添加到结果中

► 导航类方法如下表所示

方 法	说 明
<code>splash:go</code>	加载一个 URL 到浏览器
<code>splash:set_content</code>	加载指定的内容（通常是 HTML）到浏览器
<code>splash:lock_navigation</code>	锁定导航状态
<code>splash:unlock_navigation</code>	解锁导航状态
<code>splash:set_user_agent</code>	设置请求头内的 User-Agent
<code>splash:set_custom_headers</code>	设置自定义请求头
<code>splash:on_request</code>	允许过滤或替换相关资源的请求，它还允许对每个请求设置 HTTP 或 Socks5 代理服务器
<code>splash:on_response_headers</code>	允许基于请求头（例如，基于 Content-Type）来过滤请求
<code>splash:init_cookies</code>	初始化 Cookie 对象
<code>splash:add_cookie</code>	向当前请求添加 Cookie
<code>splash:get_cookies</code>	获取指定的 Cookie 对象
<code>splash:clear_cookies</code>	清除所有的 Cookie
<code>splash:delete_cookies</code>	删除一个或多个 Cookie

► 延时类方法如下表所示

方 法	说 明
<code>splash:wait allows</code>	允许等待指定的数量的时间
<code>splash:call_later</code>	允许将任务安排至将来执行

续表

方 法	说 明
splash:wait_for_resume	允许等待至某个 JS 事件被触发
splash:with_timeout	允许限制在代码块中花费的时间

➤ 提取页面信息类方法如下表所示

方 法	说 明
splash:html	返回页面 HTML 内容
splash:url	返回浏览器当前加载的 URL
splash:evaljs	运行 JS 脚本并返回数据
splash:jsfunc	同上
splash:select	在页面运行 CSS 选择器返回元素对象
splash:select_all	在页面运行 CSS 选择器返回所有与之匹配的元素对象
element:text	返回 DOM 元素对象内的文字
element:bounds	返回一个元素的 BOX 模型的边界
element:styles	返回元素被最终的样式
element:form_values	返回<form>元素的值的集合

➤ 屏幕截取类方法如下表所示

方 法	说 明
splash:png	截取当前屏幕并保存为 PNG 格式
splash:jpeg	截取当前屏幕并保存为 JPEG 格式
splash:set_viewport_full	将当前视点（截取屏幕的可视区域）设置为全屏
splash:set_viewport_size	更改视点大小
element:png	将指定元素的显示区域截取并保存为 PNG 图像
element:jpeg	将指定元素的显示区域截取并保存为图像

➤ 交互类方法如下表所示

方 法	说 明
splash:runjs	运行 JavaScript 脚本但不返回结果
splash:evaljs	运行 JavaScript 脚本并返回结果
splash:jsfunc	执行一个定义的 JavaScript 函数
splash:autoload	允许在每个网页开始渲染时预先载入 JavaScript 脚本库或者执行一些 JavaScript 代码
splash:mouse_click	模拟并触发鼠标单击网页的事件
splash:mouse_hover	模拟并触发鼠标悬停在网页上的事件
splash:mouse_press	模拟并触发鼠标点击网页按下时事件



续表

方 法	说 明
<code>splash:mouse_release</code>	模拟并触发鼠标在网页上单击后释放的事件
<code>element:mouse_hover</code>	模拟并触发鼠标在元素上掠过的事件
<code>splash:send_keys</code>	模拟在网页上敲击键盘上某个指定的键
<code>splash:send_text</code>	模拟在网页上用键盘输入文字
<code>element:send_keys</code>	模拟在元素上敲击键盘上某个指定的键
<code>element:send_text</code>	模拟在元素上敲击键盘上某个指定的键。能用 <code>element:form_values</code> 初始化表单内的输入字段的值, 或者用 <code>element:fill</code> 方法对值进行更新, 或者用 <code>element:submit</code> 提交表单
<code>splash.scroll_position</code>	将网页滑动至指定位置

➤ 生成请求类方法如下表所示

方 法	说 明
<code>splash:http_get</code>	直接发送一个 HTTP GET 请求, 并获得一个响应, 而不需要加载页面到浏览器
<code>splash:http_post</code>	发送一个 HTTP POST 请求并获得一个响应, 而不会把页面加载到浏览器
<code>splash:har</code>	以 HAR 格式返回所有请求和响应
<code>splash:history</code>	返回有关重定向和加载到主浏览器窗口的页面的历史信息
<code>splash:on_request</code>	在发起网页请求时触发该事件
<code>splash:on_response_headers</code>	当获得完整响应头时触发该事件
<code>splash:on_response</code>	当获取完整的原生响应对象时触发该事件
<code>splash.response_body_enabled</code>	在 <code>splash:har</code> 属性与 <code>splash:on_response</code> 事件内写入完整的网页正文

➤ 浏览选项如下表所示

属 性	说 明
<code>splash.js_enabled</code>	是否开启 JavaScript 解释支持
<code>splash.private_mode_enabled</code>	是否关闭私密访问模式, 某些网站需要打开此开关, 因为 Webkit 在私密模式下 <code>localStorage</code> 将不可用
<code>splash.images_enabled</code>	是否启用自动图片下载
<code>splash.plugins_enabled</code>	是否启用插件功能, 如 Flash
<code>splash.resource_timeout</code>	允许在超时之后放慢或挂起请求到的相关资源

### 4.5.3.5 常用技巧

接下来介绍一些 scrapy-splash 在虫术中的常用技巧。

#### 异步加载

首先需要通过一个实际的例子来演示如何使用 scrapy-splash，这里选择爬取某电商首页的异步加载内容。

打开首页时只会将导航菜单加载出来，其他具体内容都是异步加载的，下图有个“排行榜”内容也是通过 JavaScript 异步加载的。现在通过爬取这个“排行榜”三个字来说明普通的 Scrapy 爬取和通过使用 Splash 加载异步内容的区别。



首先写一个简单的测试 Spider，不使用 Splash。排行榜的 XPath 定位是 `//div[@id="J_top"]/div/a/h3/text()`，具体的代码如下所示。

```
class JDSyncSpider(scrapy.Spider):
    name = "xx-sync"
    allowed_domains = ["xx.com"]
    start_urls = [
        "http://www.xx.com/2017"
    ]

    def parse(self, response):
        logging.info(u'-----直接获取首页测试-----')
        special = response.xpath('//div[@id="J_top"]/div/a/h3/text()').extract_first()
        logging.info(u'find: %s' % special)
        logging.info(u'-----success-----')
```

运行结果:

```
2018-01-01 15:37:57 [scrapy.core.engine] DEBUG: Crawled (200) <GET
https://www.xx.com/2017> (referer: None)
2018-01-01 15:37:58 [root] INFO: -----直接获取首页测试-----
2018-01-01 15:37:58 [root] INFO: find: None
```



```
2018-01-01 15:37:58 [root] INFO: -----success-----
2018-01-01 15:37:58 [scrapy.core.engine] INFO: Closing spider (finished)
```

找不到“排行榜”三个字。

接下来使用 Splash 来爬取，具体代码如下：

```
class JDAsyncSpider(scrapy.Spider):
    name = "xx-async"
    allowed_domains = ["xx.com"]
    start_urls = [
        "http://www.xx.com/2017"
    ]

    def start_requests(self):
        splash_args = {
            'wait': 0.5,
        }
        for url in self.start_urls:
            yield SplashRequest(url, self.parse,
                               endpoint='render.html',
                               args=splash_args)

    def parse(self, response):
        logging.info(u'-----使用 splash 爬取首页异步加载内容-----')
        special = response.xpath('//div[@id="J_top"]/div/a/h3/text()').
extract_first()
        logging.info(u"find: %s" % special)
        logging.info(u'-----success-----')
```

运行结果：

```
2018-01-01 15:39:01 [scrapy.core.engine] DEBUG: Crawled (200) <GET
http://www.xx.com/2017 via http://localhost:8050/render.html> (referer: None)
2018-01-01 15:39:01 [root] INFO: -----使用 splash 爬取首页异步加载内容-----
2018-01-01 15:39:01 [root] INFO: find: 排行榜
2018-01-01 15:39:01 [root] INFO: -----success-----
2018-01-01 15:39:01 [scrapy.core.engine] INFO: Closing spider (finished)
```

可以看出已经找到了“排行榜”，说明异步加载内容爬取成功！

### 网页内容与截屏同时加载

Splash 有一个比较有趣的用法，就是使用 `render.json` 的服务节点，通过参数控制 Splash，同时加载 HTML 内容并对当前的窗口进行截屏，只要在 `splash_args` 参数中将 `html`、`png`、`render_all` 同时设置为 1（代表 True）即可，具体做法如下：

```
import json
import base64
import scrapy
from scrapy_splash import SplashRequest

class JDContentAndScreenSpider(scrapy.Spider):
    name = "xx-async"
    allowed_domains = ["xx.com"]
    start_urls = [
        "http://www.xx.com/2017"
    ]

    def start_requests(self):
        splash_args = {
            'html': 1,
            'png': 1,
            'width': 600,
            'render_all': 1
        }
        for url in self.start_urls:
            yield SplashRequest(url, self.parse,
                               endpoint='render.json',
                               args=splash_args)

    def parse(self, response):
        html = response.body
        title = response.css('title').extract_first()
        png_bytes = base64.b64decode(response.data['png'])
```

### 向Lua脚本传递参数

某些情况下，我们可能需要从蜘蛛向 Splash 上运行的 Lua 脚本传递一些运行参数，这个听



起来挺复杂的问题其实在 scrapy-splash 中的处理也是相当简单的,“万能”的 args 参数也可以解决这一切。

只要向 args 设置的所有非 scrapy-splash 保留的参数都会被传递进入到 Lua 脚本的执行进程中,例如:

```
import json
import base64
from scrapy_splash import SplashRequest

script = """
-- 参数:
-- * url - URL
-- * css - CSS 选择器
-- * pad - 截屏的留白尺寸

-- 此函数用于向元素添加留白
function pad(r, pad)
    return {r[1]-pad, r[2]-pad, r[3]+pad, r[4]+pad}
end

-- main 函数
function main(splash)

    -- 此函数将返回元素的盒子边界
    local get_bbox = splash:jsfunc([[
        function(css) {
            var el = document.querySelector(css);
            var r = el.getBoundingClientRect();
            return [r.left, r.top, r.right, r.bottom];
        }
    ]])

    assert(splash:go(splash.args.url))
    assert(splash:wait(0.5))

    splash:set_viewport_full()
```

```

    local region = pad(get_bbox(splash.args.css), splash.args.pad)
    return splash.png{region=region}
end
"""

class MySpider(scrapy.Spider):

    # ...
    yield SplashRequest(url, self.parse_element_screenshot,
                        endpoint='execute',
                        args={
                            'lua_source': script,
                            'pad': 32,
                            'css': 'a.title'
                        })

    # ...
    def parse_element_screenshot(self, response):
        image_data = response.body
        # ...

```

在生成 `SplashRequest` 请求对象时，由 Python 设置的 `args` 可以在 Lua 的 `main` 函数中通过访问 `splash:args` 字典重新获得。

接下来，我们将这个方向反过来，从 Lua 端将处理后的数据发送给蜘蛛，也就是在 `main` 函数中返回一个 Lua table，然后就能从 `response` 对象中重新将它们读取出来。需要注意的一点是，此时使用的服务终结点是 `/execute` 而不再是 `render.xxx`。

```

import scrapy
from scrapy_splash import SplashRequest

script = """
function main(splash)
    splash:init_cookies(splash.args.cookies)
    assert(splash:go{
        splash.args.url,
        headers=splash.args.headers,
        http_method=splash.args.http_method,

```



```

        body=splash.args.body,
    })
    assert(splash.wait(0.5))

    local entries = splash:history()
    local last_response = entries[#entries].response
    return {
        url = splash:url(),
        headers = last_response.headers,
        http_status = last_response.status,
        cookies = splash:get_cookies(),
        html = splash:html(),
    }
end
"""

class MySpider(scrapy.Spider):

    # ...
    yield SplashRequest(url, self.parse_result,
        endpoint='execute',
        cache_args=['lua_source'],
        args={'lua_source': script},
        headers={'X-My-Header': 'value'},
    )

    def parse_result(self, response):
        # 此处就可以从 response 中读取 Lua 中返回的 JSON
        pass

```

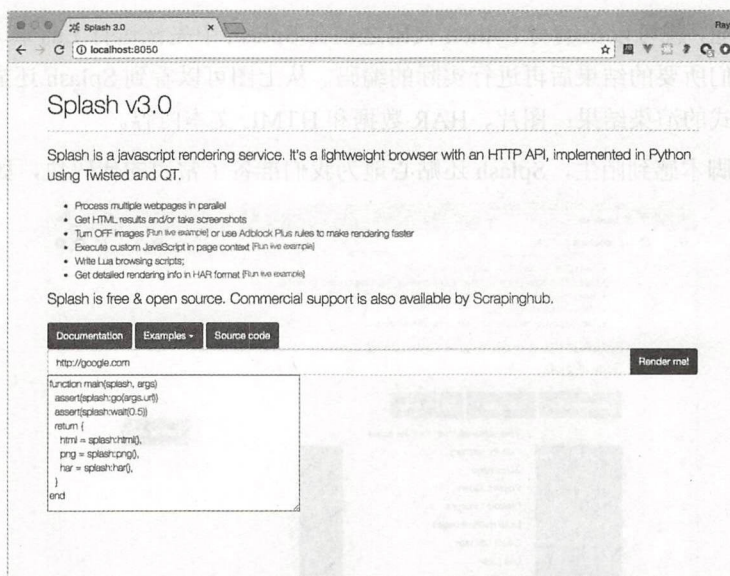
## Splash的Web工具

如果前面介绍的 Splash 内容令你觉得一时很难掌握,那么接下来可以使用一种更加简单的技巧,这就是 Splash 3.0 提供的 Web 工具。在了解了以上理论的基础上来使用这个 Web 工具,很多的困惑与疑问都可以迎刃而解。

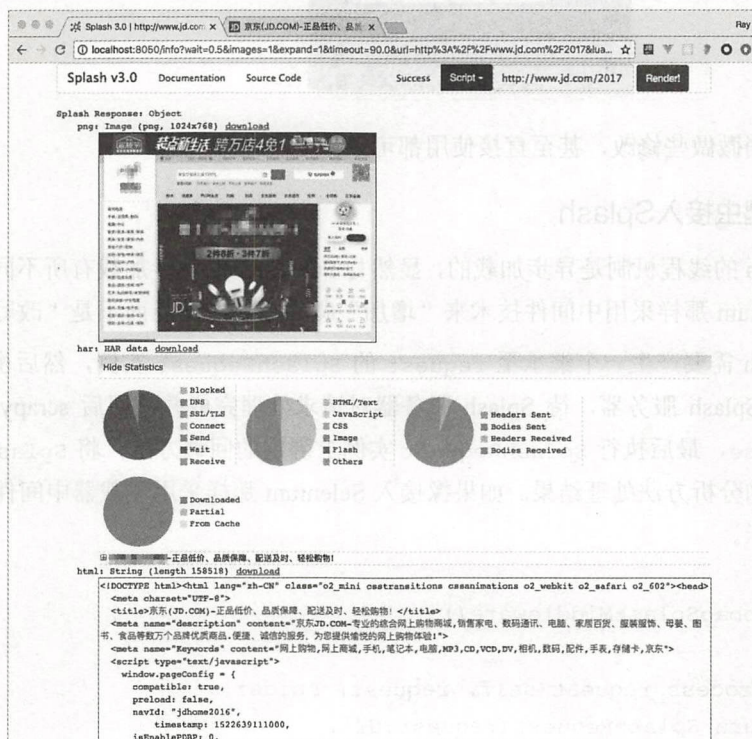
运行 Splash Web 工具的方法具体如下:

- (1) 启动安装有 Splash 的 Docker 实例。
- (2) 在浏览器打开 Splash 的 Web 地址(默认为 <http://localhost:8050>)。

然后就会看到以下的使用界面:



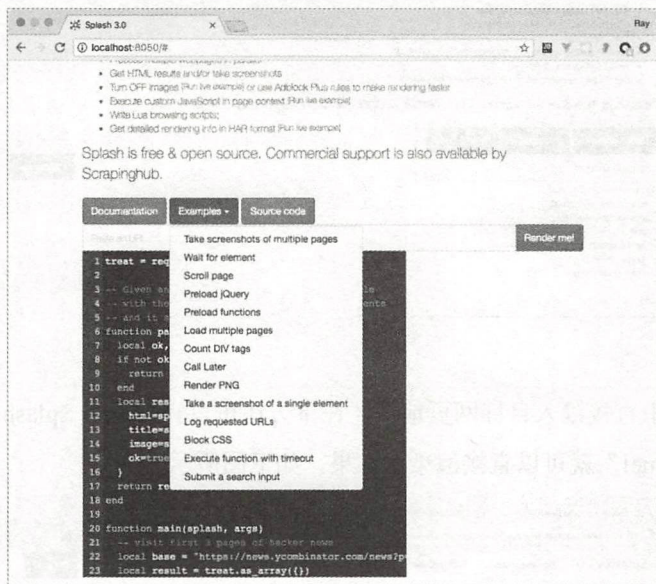
可以在输入框中直接键入目标网页地址，在下方还可以直接编写 Splash 的 Lua 脚本。单击“Render me!”就可以直接渲染出结果，如下图所示。





通过这个界面,就可以在编写 Python 代码之前在 Splash 中先试运行 Lua 脚本,以确认脚本能正常渲染出我们所要的结果后再进行实际的编码。从上图可以看到 Splash 还显示了网页的加载状况及各种格式的渲染结果:图片、HAR 数据和 HTML 文本内容。

如果对 Lua 脚本感到陌生, Splash 还贴心地为我们准备了常用的代码块,如下图所示。



在源码内稍微做些修改,甚至直接使用都可以。

#### 4.5.3.6 将爬虫接入 Splash

由于 Splash 的线程机制是异步加载的,显然与 Selenium 的同步加载有所不同,但我们仍然能像接入 Selenium 那样采用中间件技术来“增加”爬虫系统的功能而不是“改写”。

接入 Splash 需要产生一个继承至 request 的 SplashRequest 实例,然后由 scrapy-splash 中间件发送至 Splash 服务器,待 Splash 服务器对请求处理完成并返回后 scrapy-splash 再生成 SplashResponse,最后执行 SplashRequest 实例上绑定的回调方法,将 SplashResponse 实例传递给蜘蛛的分析方法处理结果。如果像接入 Selenium 那样采用下载器中间件,则具体的写法应该如下所示。

```
class TaobaoSplashMiddleware():

    def process_request(self, request, spider):
        return SplashRequest(request.url,
```

```

        request.callback,
        endpoint='execute',
        meta=dict(r.meta),
        args={
            'lua_source': self.lua_source,
            'wait': 3
        }
    )

```

由于 scrapy-splash 内置的 SplashMiddleware 会发出与 TBSplashMiddleware 同样的 URL, 这样会触发 SplashAwareDupeFilter 把其中的一个 URL 给过滤掉, 从而导致无法向 Splash 服务器发出正常请求。因此, 在这个示例中我们就不能采用下载器中间件而应该采用蜘蛛中间件, 在 request 还没有被发送至下载器中间处理链之前将其转换为 SplashRequest 实例, 具体代码如下所示。

```

from scrapy import signals
from scrapy_splash import SplashRequest

class TaobaoSplashSpiderMiddleware(object):

    lua_source = ""

    def process_start_requests(self, start_requests, spider):

        for r in start_requests:
            yield SplashRequest(r.url,
                                r.callback,
                                endpoint='execute',
                                meta=dict(r.meta),
                                args={
                                    'lua_source': self.lua_source,
                                    'wait': 3
                                }
            )

```

由于我们需要将页面加载到 Splash 服务器上并等待其页面上的 JavaScript 执行完成才能得到真正的结果页, 如何在 scrapy-splash 中实现如同 Selenium 那样的等待事件使得我们确切地知



道需要的元素已加载完成呢？此时就需要使用 Lua 脚本了。

在“Splash 脚本”中我们都知道了 Splash 脚本是通过一个 main 函数作为程序入口的，按前文的做法这个 main 函数的写法应该如下所示。

```
function main(splash,args)
    assert(splash:go(args.url))
    splash:wait(args.wait)
    return splash:html()
end
```

虽然这个 main 函数会让 Splash 等待一个指定等待时间，但这样做非常笼统而且不准确。由于 Splash 并没有实现这样的一个等待函数，那么我们就需要编写一个等待函数来取代 splash:wait(args.wait) 以确切地知道某个元素被 Splash 正确地渲染了。

最简单的实现思路是在 Splash 所渲染的当前页面内使用 JavaScript 来查询指定的元素是否存在。这个 JavaScript 脚本的实现逻辑如下：

- (1) 定义一个时间循环 setTimeout，确保 JavaScript 能在间隔时间内循环工作。
- (2) 假如元素被检测到，则从 JavaScript 进程退出，返回至原 Splash 执行进程。
- (3) 假如元素没有被检测到而且还在指定的重试范围内，则进入下一循环。
- (4) 假如在指定重试次数内都没有成功检测到指定元素，则直接退出至 Splash 的执行进程。

具体实现如下：

```
function main(splash) {

    var selector = '%s';
    var maxwait = %s;
    var end = Date.now() + maxwait*1000;

    function check() {
        if(document.querySelector(selector)) {
            splash.resume('Element found');
        } else if(Date.now() >= end) {
            var err = 'Timeout waiting for element';
            splash.error(err + " " + selector);
        } else {
            setTimeout(check, 200);
        }
    }
}
```

```

    }
}

check();
}

```

Splash 要执行一个 JavaScript, 必须有一个 main 函数作为程序入口, 且带有一个 splash 参数作为 Lua 与 JavaScript 两个不同进程之间的通信上下文, splash.resume 与 splash.error 都将会终止当前的 JavaScript 进程并返回至 Splash 的 Lua 进程中。

在 Splash 的 Lua 中则使用 wait\_for\_resume 函数来调用 JavaScript 脚本, 将两种语言的互调用合二为一会得到以下的 wait\_for\_element 函数:

```

function wait_for_element(splash, css, maxwait)
    if maxwait == nil then
        maxwait = 10
    end

    return splash:wait_for_resume(string.format([[
        function main(splash) {
            var selector = '%s';
            var maxwait = %s;
            var end = Date.now() + maxwait*1000;

            function check() {
                if(document.querySelector(selector)) {
                    splash.resume('Element found');
                } else if(Date.now() >= end) {
                    var err = 'Timeout waiting for element';
                    splash.error(err + " " + selector);
                } else {
                    setTimeout(check, 200);
                }
            }

            check();
        }
    ]],css, maxwait))

```

End



最后将 Lua 的 main 函数中的 wait 改写为 wait\_for\_element 则完成了 Lua 脚本的编写:

```
function main(splash, args)
    splash:go(args.url)
    wait_for_element(splash, "#mainsrp-itemlist")
    return splash:html()
end
```

以下是 TBSplashSpiderMiddleware 的完整代码:

```
from scrapy import signals
from scrapy_splash import SplashRequest

class TBSplashSpiderMiddleware(object):

    lua_source = """
        function wait_for_element(splash, css, maxwait)
            -- Wait until a selector matches an element
            -- in the page. Return an error if waited more
            -- than maxwait seconds.
            if maxwait == nil then
                maxwait = 10
            end
            return splash:wait_for_resume(string.format([[
                function main(splash) {
                    var selector = '%s';
                    var maxwait = %s;
                    var end = Date.now() + maxwait*1000;

                    function check() {
                        if(document.querySelector(selector)) {
                            splash.resume('Element found');
                        } else if(Date.now() >= end) {
                            var err = 'Timeout waiting for element';
                            splash.error(err + " " + selector);
                        } else {

```

```

        setTimeout(check, 200);
    }
}
check();
}
]], css, maxwait))
end

function main(splash, args)
    splash:go(args.url)
    wait_for_element(splash, "#mainsrp-itemlist")
    return splash:html()
end
"""

```

```
def process_start_requests(self, start_requests, spider):
```

```
    for r in start_requests:
```

```
        yield SplashRequest(r.url,
```

```
                               r.callback,
```

```
                               endpoint='execute',
```

```
                               meta=dict(r.meta),
```

```
                               args={ 'lua_source': self.lua_source }

```

```
        )

```

按前文所述配置 settings.py 配置文件来启用 scrapy-splash 与激活 TBSplashSpider-Middleware 中间件, settings.py 的完整代码如下所示。

```
# -*- coding: utf-8 -*-
```

```
BOT_NAME = 'TB_splash'
```

```
SPLASH_URL = 'http://localhost:8050'
```

```
SPIDER_MODULES = ['TB_splash.spiders']
```

```
NEWSPIDER_MODULE = 'TB_splash.spiders'
```



```

USER_AGENT = 'Mozilla/5.0 (Windows NT 10.0; WOW64; rv:51.0) Gecko/20100101
Firefox/51.0'

# Obey robots.txt rules
# 必须配置为假, 因为该网站的 robot 上声明产品搜索页是不欢迎被蜘蛛爬取的
ROBOTSTXT_OBEY = False

DOWNLOADER_MIDDLEWARES = {
    'scrapy_splash.SplashCookiesMiddleware': 723,
    'scrapy_splash.SplashMiddleware': 725,
    'scrapy.downloadermiddlewares.httpcompression.HttpCompression-
Middleware': 810,
}

DUPEFILTER_CLASS = 'scrapy_splash.SplashAwareDupeFilter'

```

## 4.6 数据存储与后处理

当蜘蛛完成了对数据的爬取工作后, 就需要将数据暂时或永久性地保存起来。其实这一步骤在“Scrapy 基础”的“管道”一节中也有所提及。Scrapy 的管道非常适合作为数据存储与数据后续处理的扩展区域。因为管道本身有着“无限”扩展的能力, 不同的管道相互拼接实现了形形色色的后处理功能。

我们一起来回顾一下在“管道”一节中曾列举的一个用于将 Item 存储到 JSON 文件中的管道代码示例:

```

import json

class JsonFeedPipeline:

    def __init__(self):
        self.json_file = open('feed.json', 'wt')

    def process_item(self, item, spider):
        line = json.dumps(dict(item)) + "\n"
        json_file.write_line(line)

```



运用虫术的过程中绝不只遇到将数据保存为 JSON 一种情况，不同的需求自然会诞生对特定文件格式的需要。因此，我们可能还需要将 Item 数据保存到数据库或各种文件中，甚至下载其中的图像或者视频流。

### 4.6.1 图片的下载与存储

Scrapy 提供了一个专用图片管道 `ImagesPipeline` 来下载属于某个特定数据项目中的图片。以下是 `ImagesPipeline` 类的一些使用特性与能力：

- (1) 将所有下载的图片转换成通用的格式 (JPG) 和模式 (RGB)。
- (2) 避免重新下载最近已经下载过的图片。
- (3) 自动生成缩略图并保存成文件。
- (4) 检测图像的宽/高，过滤不满足限制的图片。
- (5) 这个管道也会为当前安排好要下载的图片保留一个内部队列，并将到达的包含相同图片的项目连接到这个队列中。这样可以避免多次下载几个项目共享的同一张图片。

`ImagesPipeline` 通过 `Pillow` 来生成缩略图，并将图片归一化为 JPEG/RGB 格式。因此，为了使用图片管道，需要安装这个库。`Pillow` 在前文中已有提及，在此不再赘述。

#### 使用图片管道

Scrapy 内置的 `ImagesPipeline` 使用起来非常简单，具体有两种做法：第一种是通过配置直接使用 `ImagesPipeline`；第二种是继承 `ImagesPipeline` 改写存储逻辑。

先来看看 `ImagesPipeline` 的典型工作流程，了解一下它是如何运作的：

- (1) 在一个蜘蛛里抓取一个 Item，并把其中图片的 URL 放入 `image_urls` 属性中。
- (2) Item 在蜘蛛内返回，进入 Item 管道。
- (3) 当 Item 进入 `ImagesPipeline` 时，`image_urls` 组内的 URLs 将被 Scrapy 的调度器和下载器安排下载，当优先级更高时，会在其他页面被抓取前处理。项目会在这个特定的管道阶段保持“locker”的状态，直到完成图片的下载（或者由于某些原因未完成下载）。
- (4) 当图片下载完，另一个组 (`images`) 将被更新到结构中。这个组将包含一个字典列表，其中包括下载图片的信息，比如下载路径、源抓取地址（从 `image_urls` 属性获得）和图片的校验码。`images` 列表中的图片顺序将和源 `image_urls` 组保持一致。如果某个图片下载失败，则记录下错误信息，图片也不会出现在 `images` 组中。

为了使用图片管道，需要在自定义的 Item 类中加入 `image_urls` 和 `images` 属性：





```
import scrapy

class MyItem(scrapy.Item):

    # ... other item fields ...
    image_urls = scrapy.Field()
    images = scrapy.Field()
```

然后在配置文件 `settings.py` 中添加 `ITEM_PIPELINES` 配置项并指定为 `ImagesPipeline`，具体如下所示。

```
# settings.py
ITEM_PIPELINES = {'scrapy.pipelines.images.ImagesPipeline': 1}
```

最后将 `IMAGES_STORE` 配置项指定为一个有效的文件夹，用来存储下载的图片，否则管道将保持禁用状态。

例如：

```
# settings.py
IMAGES_STORE = '/Users/Ray/projects/my_crawler/images/'
ITEM_PIPELINES = {'scrapy.contrib.pipeline.images.ImagesPipeline': 1}
```

完成以上配置后，通过 `$ scrapy crawl` 指令运行爬虫就会自动将图片保存到 `/Users/Ray/projects/my_crawler/images/` 中了。

## 图片存储

图片会使用它们 URL 的 SHA1 Hash 作为文件名。比如，对下面的图片 URL：

```
http://www.example.com/image.jpg
```

它的 SHA1 Hash 值为：

```
3afec3b4765f8f0a07b78f98c07b83f013567a0a
```

当文件下载完成后，会被保存成为下面的文件：

```
<IMAGES_STORE>/full/3afec3b4765f8f0a07b78f98c07b83f013567a0a.jpg
```



其中：

- `<IMAGES_STORE>`是定义在 `IMAGES_STORE` 设置中的文件夹路径。
- `full` 是用来区分图片和缩略图（如果使用）的一个子文件夹。

## 图片的有效期

图像管道用来避免下载最近已经下载的图片。使用 `IMAGES_EXPIRES` 配置项调整失效期限，可以用天数来指定，具体如下所示。

```
# 90 天的图片失效期限
IMAGES_EXPIRES = 90
```

当图片失效时，Scrapy 就会重新下载该图片，反之则会跳过下载过程。

## ➤ 缩略图生成

图片管道可以自动创建下载图片的缩略图。为了使用这个特性，需要设置 `IMAGES_THUMBS` 配置项字典，其关键字为缩略图名字，值为它们的大小尺寸。

比如：

```
IMAGES_THUMBS = {
    'small': (50, 50),
    'big': (270, 270),
}
```

当使用这个特性时，图片管道将使用下面的格式来创建各个特定尺寸的缩略图：

```
<IMAGES_STORE>/thumbs/<size_name>/<image_id>.jpg
```

其中：

- `<size_name>`是 `IMAGES_THUMBS` 字典的关键字（`small`、`big` 等）。
- `<image_id>`是图像 URL 的 SHA1 Hash。

例如，使用 `small` 和 `big` 缩略图名字的图片文件：

```
<IMAGES_STORE>/full/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/small/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/big/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
```





第一个是从网站下载的完整图片。

### 过滤小图片

在 `IMAGES_MIN_HEIGHT` 和 `IMAGES_MIN_WIDTH` 设置中指定最小允许的尺寸就可以过滤那些太小的图片。

比如:

```
IMAGES_MIN_HEIGHT = 110
IMAGES_MIN_WIDTH = 110
```

**注意:** 这些尺寸一点也不影响缩略图的生成。

默认情况下 `ImagesPipeline` 是没有尺寸限制的, 因此所有图片都将被处理。

### 自定义图片管道

自定义图片管道最简单的方法就是继承自 `ImagesPipeline` 类, 以下是它的类定义:

```
class scrapy.pipelines.images.ImagesPipeline
```

在上面的工作流程中可以看到, 管道会得到图片的 `URL` 并从项目中下载。需要重写 `get_media_requests()` 方法, 并对各个图片 `URL` 返回一个 `request`:

```
def get_media_requests(self, item, info):
    for image_url in item['image_urls']:
        yield scrapy.Request(image_url)
```

当然上述写法就是 `ImagesPipeline` 原有 `get_media_requests` 的代码, 从中我们就可以了解为什么要在自定义的 `Item` 中定义 `image_urls`。当我们遇到更复杂的情况时就可通过改写 `get_media_requests` 以产生图片下载请求。

如果 `get_media_requests()` 方法返回 `None`, 则意味着项目中没有图片可下载。

这些请求将被管道处理, 当它们完成下载后, 会以双元素元组列表形式传送到 `item_completed(results, items, info)` 方法的 `results` 参数中。

以下为 `results` 参数中具有的关键及其顺序:

- `success` 是一个布尔值, 当图片成功下载时为 `True`, 因为某个原因下载失败时为 `False`。



- `image_info_or_error` 是一个包含下列关键字的字典（如果成功则为 `True`，出问题时为 `Twisted Failure`）。
- `url`——图片下载的 URL。这是从 `get_media_requests()` 方法返回请求的 URL。
- `path`——图片存储的路径（类似 `IMAGES_STORE`）。
- `checksum`——图片内容的 MD5 Hash。

下面是 `results` 参数的一个典型值：

```
[ (True,
  {'checksum': '2b00042f7481c7b056c4b410d28f33cf',
   'path': 'full/7d97e98f8af710c7e7fe703abc8f639e0ee507c4.jpg',
   'url': 'http://www.example.com/images/product1.jpg'}),
  (True,
   {'checksum': 'b9628c4ab9b595f72f280b90c4fd093d',
    'path': 'full/1ca5879492b8fd606df1964ea3c1e2f4520f076f.jpg',
    'url': 'http://www.example.com/images/product2.jpg'}),
  (False,
   Failure(...)) ]
```

当一个单独项目中的所有图片请求完成时（要么完成下载，要么因为某种原因下载失败），`ImagesPipeline.item_completed()` 方法将被调用。`item_completed()` 方法需要返回一个输出，其将被送到随后的 `Item` 管道中，因此需要返回（或者丢弃 `Item`，如同在任意管道中所做的一样）。

以下是一个 `item_completed()` 方法的例子，其中将下载的图片路径（传入 `results` 中）存储到 `image_paths` 项目组中，如果没有图片则将其丢弃：

```
from scrapy.exceptions import DropItem

def item_completed(self, results, item, info):
    image_paths = [x['path'] for ok, x in results if ok]
    if not image_paths:
        raise DropItem(u'没有图片')
    item['image_paths'] = image_paths
    return item
```

下面是一个图片管道的完整例子，其方法如上所示。





```
import scrapy
from scrapy.pipelines.images import ImagesPipeline
from scrapy.exceptions import DropItem

class MyImagesPipeline(ImagesPipeline):

    def get_media_requests(self, item, info):
        for image_url in item['image_urls']:
            yield scrapy.Request(image_url)

    def item_completed(self, results, item, info):
        image_paths = [x['path'] for ok, x in results if ok]
        if not image_paths:
            raise DropItem(u'没有图片')

        item['image_paths'] = image_paths
        return item
```

## 4.6.2 示例：产品图片采集

本节为前面编写的爬虫增加一个产品图片采集功能，根据在上一节介绍的编写图片下载管道的办法来编写产品图片下载管道并将下载后的图片地址保存至 Item 的 image\_path 中，具体代码如下所示。

```
import scrapy
from scrapy.pipelines.images import ImagesPipeline
from scrapy.exceptions import DropItem

class ProductImagePipeline(ImagesPipeline):

    def get_media_requests(self, item, info):
        for image_url in item['image_urls']:
            yield scrapy.Request(image_url[2:])

    def item_completed(self, results, item, info):
        image_paths = [x['path'] for ok, x in results if ok]
        if not image_paths:
```



```

        raise DropItem(u'没有图片')
    item['image_path'] = image_path
    return item

```

产品搜索页搜索到的图片都比较小，所以我们并不需要生成缩略图，在配置文件 `settings.py` 中加入以下配置项以启用管道：

```

IMAGES_STORE = '~/taobao/images/'
ITEM_PIPELINES = {'taobao.pipelines.ProductImagePipeline': 1}
IMAGES_MIN_HEIGHT = 250
IMAGES_MIN_WIDTH = 250

```

### 4.6.3 导出到数据文件

实现爬虫时经常提到的需求就是能合适地保存爬取的数据，或者说生成一个带有爬取数据的“导出文件”（通常称为“Feed Exporter”）来供其他系统使用。

Scrapy 自带了 Feed 输出，并且支持多种序列化格式（serialization format）及存储方式（storage backends）。

当抓取了需要的数据（Items）后，需要将数据持久化或导出数据，并应用在其他程序中。这是整个抓取过程的目的。

为此，Scrapy 提供了 Item Exporters 来创建不同的输出格式，如 XML、CSV 或 JSON。

Feed Exporter 的本质是什么呢？由 Feed Exporter 的定义不难理解，FeedExporter 实际上就是一个序列化器，也就是将对象实例转化并保存到某一指定的可储格式的文件中。序列化可以说是面向对象语言中必备的功能，一般意义上的序列化只是将对象实例与类型信息直接转化为二进制流的形式，这样就能将对象的“状态”存储起来，以后用的时候再通过反序列化将其从文件中恢复成对象实例，这个操作有点像科幻片中人类被冰冻与解冻的过程。

然而，如果是基于二进制方式的序列化，则会导致保存的文件在反序列化时要依赖于被序列化时的对象类型。也就是说，这个文件必须依赖于对象被序列化时所采用的语言。这样做就没有任何开放性，文件是不能被共享与兼容的。这也正是 JSON 格式能作为现代互联网中最通用的一种序列化对象的承载文件的原因，因为所有的语言都支持它，不同语言间的对象都可以互相序列化与反序列化。说得更明白一些，就是在 Java 中生成的一系列对象实例可以在 Node.js 中被重新还原为 JavaScript 上下文中的对象。当然所有的通用文件格式都可以具有这样的特性，比如说 XML，只是 XML 的文件格式太过严格，导致文件尺寸变得极为庞大，并不利于在网络





上高速传递。

Feed Exporter 就是将 Scrapy 的蜘蛛所生成的 Item 对象序列化为各种格式的文件的一个模块。Feed Exporter 只是一种统称，在 Scrapy 官方文档中都非常混乱，实际上用的只是 ItemExporter 而已。

## 怎样使用

Feed Exporter 只需要在 Scrapy 的配置文件 (settings.py) 中进行简单的声明即可应用。Scrapy 内置了以下几种 Feed Exporter，如下表所示。

导出格式	类	说明
json	JsonItemExporter	输出 JSON 文件格式，所有对象将写进一个对象的列表
jsonlines	JsonLinesItemExporter	输出 JSON 文件格式，每行写一个 JSON-encoded 项
xml	XmlItemExporter	输出 XML 文件格式
csv	CsvItemExporter	输出 CSV 文件格式
pickle	PickleItemExporter	输出至 Python 通用的对象结构序列化格式
marshal	MarshalItemExporter	输出至 Python 通用的对象结构序列化格式（兼容旧版本格式，主要用于 .pyc 文件）

在 settings.py 文件中加入 FEED\_FORMAT 配置项来设置默认的序列化格式，如下所示。

```
# settings.py
FEED_FORMAT = 'json'
```

为了使用 Item Exporter，必须对 Item Exporter 及其参数 (args) 实例化。每个 Item Exporter 需要不同的参数。在实例化 exporter 之后，必须：

- (1) 调用 start\_exporting() 方法以表示 exporting 过程的开始。
- (2) 对要导出的每个项目调用 export\_item() 方法。
- (3) 调用 finish\_exporting() 以表示 exporting 过程的结束。

我们必须了解一点：Item Exporter 实质上是管道中的一个扩展处理，因此要使用 Item Exporter 就必须在管道中实例化它。在下面的代码中，可以看到一个 XmlExportPipeline，它使用 Item Exporter 导出 Items 到不同的文件中：

```
from scrapy import signals
from scrapy.contrib.exporter import XmlItemExporter
```



```

class XmlExportPipeline(object):

    def __init__(self):
        self.files = {}

    @classmethod
    def from_crawler(cls, crawler):
        pipeline = cls()
        crawler.signals.connect(pipeline.spider_opened, signals.spider_opened)
        crawler.signals.connect(pipeline.spider_closed, signals.spider_closed)
        return pipeline

    def spider_opened(self, spider):
        file = open('%s_products.xml' % spider.name, 'w+b')
        self.files[spider] = file
        self.exporter = XmlItemExporter(file)
        self.exporter.start_exporting()

    def spider_closed(self, spider):
        self.exporter.finish_exporting()
        file = self.files.pop(spider)
        file.close()

    def process_item(self, item, spider):
        self.exporter.export_item(item)
        return item

```

上述的 `XmlExportPipeline` 首先在类方法 `from_crawler` 中绑定 `spider_opened` 和 `spider_closed` 的事件处理方法。当蜘蛛开始爬网即 `spider_opened` 被触发时，实例化 `XmlItemExporter` 并为其指定一个目标存储文件。然后调用 `XmlItemExporter` 的 `start_exporting` 方法以通知 `XmlItemExporter` 开始处理 Items。当 `XmlItemPipeline` 的 `process_item` 方法被调用时，调用 `XmlItemExporter` 的 `exporter_item` 对当前管道所处理的 Item 进行序列化并写入文件。最后在 `spider_closed` 事件即蜘蛛完成爬取并关闭之前调用 `finish_exporting` 以通知 `XmlItemExporter` 将所有已写入导出文件的内容保存到磁盘中。

通过上述的例子我们已经了解了 `ItemExporter` 的三个基本方法：

- `start_exporting()` —— 传入文件对象并通知 `ItemExporter` 初始化其目标写入文件。





- `export_item(item)` —— 将 `Item` 序列化并写入文件。
- `finish_exporting()` —— 结束文件写入并进行保存。

### 控制Item的序列化

上文中不断提及用 `exporter_item` 方法将 `Item` 进行序列化处理，如果想精确地控制 `Item` 中某个属性的序列化过程，则又应如何处理呢？

有两种方法可以自定义一个字段如何被序列化：

(1) 在 `field` 类中声明一个 `serializer`。可以在 `field metadata` 中声明一个 `serializer`，该 `serializer` 必须可调用，并返回它的序列化形式。

```
import scrapy

def serialize_price(value):
    return '$ %s' % str(value)

class Product(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field(serializer=serialize_price)
```

(2) 覆盖 (overriding) 继承的 `ItemExporter` 的 `serialize_field()` 方法来自定义如何输出数据。

在自定义代码后确保调用父类的 `serialize_field()` 方法。

```
from scrapy.contrib.exporter import XmlItemExporter

class ProductXmlExporter(XmlItemExporter):

    def serialize_field(self, field, name, value):
        if field == 'price':
            return '$ %s' % str(value)
        return super(Product, self).serialize_field(field, name, value)
```

## 4.6.4 导出到数据库

对于暂时性的数据爬取或者非结构化的数据可以采用前文中提到的存储后端方式，毕竟这种做法投入的开发时间较少、开发成本较低。但这种方式不适合持久性的数据爬取或者数据存



储量极大的情况。再者，一旦需要对数据进行筛选、查询甚至是持久性存储，将数据存储为非结构化形式会带来诸多不便。面对这类情况时，将爬取的数据存储到数据库中是不二选择。

### 数据库的选择

数据库技术的发展已经拥有相当悠久的历史了，这当然是相对于计算机的诞生历史来说的。从文件型的桌面数据库到支持复杂数据查询的 SQL 数据库，从面向独立存储的数据库到面向超大型数据场景的分布式数据库，当下我们能选择的优秀数据库可以说是多种多样的，老一代的数据库不断地迭代升级，新生代面向大数据应用的数据库也是层出不穷。

正如开篇提到的爬虫系统，在大多数情况下处于大数据生态结构的前端，因此爬虫系统所采用的数据库一般来说都应该采用并发性强、I/O 速度快、容错性强、数据容量大的数据库为宜。

数据库的选择非常重要，因为它并不容易更换，一旦部署几乎难以更换。并不是选择功能最强大的数据库就是最好的，而是应该寻找最适合的。不同的数据应用场景、技术人员配备、用户的实际需求都可能成为我们选择数据库的重要考虑因素。本节会以常见的 SQL 数据库为例，毕竟成熟的 SQL 数据库比较多，而且具有很强的稳定性，在绝大多数的应用场景中都有它们的身影。高阶爬虫部分会介绍面向数据量更大、分布性更强的 NoSQL 数据库。

谈到 SQL 数据库，比较常用的有 Oracle、DB2、SQLServer、MySQL、SQLite、PostgreSQL 等。由于 SQLite 的处理能力很低，仅仅可以用在开发测试中，并不作为实际使用推荐。综合 Linux 和 Windows 平台上的应用，在中小型的应用场合，MySQL 和 PostgreSQL 占有相当大的应用份额；Oracle、DB2 虽然应用成本极高，但仍然在传统的商业应用中占有很高的比例；SQLServer 则只能应用于 Windows 生态中。从应用成本、部署成本和开发难度等各方面综合考虑，推荐使用 PostgreSQL，本节后面内容也会以 PostgreSQL 为主。

**注：**本节会使用 SQLAlchemy 连接与操作 PostgreSQL，还没有接触过 SQLAlchemy 的读者可以参考“SQLAlchemy 使用简介”中的内容，其中简单地介绍了 SQLAlchemy 的基本用法。

### 两种做法

Scrapy 的官方并没有给出将爬取数据存储到数据库的推荐方法，我和我的团队在实际开发运用中对此进行了多次的推敲与实践。总结出以下两种做法：

(1) 整合数据库存储管道。

(2) 独立爬取与数据库存储。

#### 整合数据库存储管道

Scrapy 的存储后端技术虽然有很强的可扩充性，但官方只给出了文件形式的存储。如果仔





细阅读上一节的内容, 则会发现存储后端仅限于文件类型使用, 不能用作数据库, 这是由 `IFeedStorage` 接口所限定的:

```
class IFeedStorage(Interface):
    """所有的存储后端口都必须实现此接口"""

    def __init__(uri):
        """通过指定的 URI 参数初始化存储"""

    def open(spider):
        """为指定的蜘蛛打开存储。此方法必须返回一个类似文件的对象, 用于数据导出"""

    def store(file):
        """存储文件流"""
```

所有的存储后端口都必须实现此接口, 如果想扩充成数据库方式, 则 `store(file)` 方法无法实现。如果强行实现则会让逻辑显得非常混乱, 将这个逻辑展开:

- (1) 通过 `ItemExporter` 将数据序列化后写入 `file` 对象。
- (2) 调用 `Storage` 的 `store(file)` 保存数据。
- (3) 从 `file` 中反序列化数据令其重新还原为数据对象。
- (4) 将数据写入数据库。

第 1 步与第 3 步如果在同一进程内, 则是毫无意义的, 但又不受我们控制, 这是由 `Scrapy` 的存储后端结构所限定的。因此我们只能通过更加上层的实现方法来实现数据库存储, 存储后端实际上是管道的一种扩展, 我们可以选择管道作为数据库存储的接入点。

首先定义配置以获取连接字符串、用户名及密码等信息, 将其保存到 `settings.py` 文件中:

```
# 数据库连接字符串
CONNECTION_STRING = 'postgresql://root:1234@localhost/spider-db'
```

如果频繁地向 `PostgreSQL` 提交变更 (`commit()`), 数据库的写入速度就会变得越来越慢, 这是由于 `PostgreSQL` 在提交时会写入数据日志 (很多数据库都有此功能)。又因为蜘蛛一次性爬取的数据量可能会非常庞大, 解决这一问题的方法是在蜘蛛开始爬网时先连接数据并产生数据库的会话上下文, 每爬取一项数据就保存当前会话上下文的数据, 当然这种保存是写入会话缓存中的。当蜘蛛完成爬网后, 利用 `PostgreSQL` 的批量写入功能一次性地将大量的数据变更存入数据库 (批量写入是一种效率极高、数据库 I/O 消耗又很低的操作) 并断开与数据库的连接



以释放资源，具体代码如下所示。

```
import database

class ArticlePostgreSQLPipeline(object):

    @classmethod
    def from_crawler(cls, crawler):
        try:
            pipeline = cls.from_settings(crawler.settings)
            crawler.signals.connect(pipeline.spider_opened, signals.spider_
opened)
            crawler.signals.connect(pipeline.spider_closed, signals.spider_
closed)
            pipeline.connection_string = crawler.settings['CONNECTION_STRING']

        except AttributeError:
            pipeline = cls()

        pipeline.crawler = crawler
        return pipeline

    def spider_opened(self, spider):
        # 连接数据库
        self.engine = create_engine(self.connection_string)
        self.session = sessionmaker(bind=self.engine)

    def process_item(self, item, spider):
        article = Article(**item)
        self.session.add(article)

    def spider_closed(self, spider):
        # 批量写入数据库
        self.session.commit()
        self.session.close()
```

#### ► 优缺点

当读懂上述代码后，一定会觉得这种办法是简单又是实际可行的。这种做法的最大优点就



是沿用了 Scrapy 的管道扩展, 代码既易读又容易配置与部署。上例只是向数据库写入一个没有依赖关系的数据表, 这种办法无疑是切实可行的。

### 独立爬取与数据库存储

但是, 当爬取的数据具有一定数据关系, 且在保存目标数据前要先向数据库添加或更改保存某项数据时, 很容易导致数据库中对某些字段的类型验证失败, 或者出现空值验证所导致的不可预知的异常。因为批量提交变更一旦失败, 所有数据项都会丢失!

如果要弥补上述的缺点, 则可以使用另一个办法: 首先采用 Scrapy 的导出文件的方式将爬取的数据暂存于磁盘, 然后开发另一个独立的程序, 专门负责将爬取的数据文件导入目标数据库中, 最后删除导出的数据文件。爬虫与数据导入是两个独立的程序, 运行在各自独立的进程中, 互不干扰。用一个命令行脚本来一次性执行它们就行了。假定数据导入程序存放在 `db_importer.py` 文件中, 这个命令行脚本如下所示。

```
# crawl_all
scrapy crawl
python db_importer.py
```

当执行 `crawl_all` 时, `db_importer.py` 必然会等待 Scrapy 完成爬网后再启动。

这种作法的优点是:

(1) 适应性强, 依赖性低——只要导出的数据文件内容格式不变, 两个程序可以独立开发、独立修改。

(2) 进程间互不干扰——即使爬取程序遇到各种不确定因素而导致局部失败(例如, 爬虫被封、网络不畅等), 或者数据导入时出现数据错误导致的异常中断, 也不会令整个爬取程序被迫中终。因为通常都是爬取程序运行完成后数据导入程序才启动。

(3) 速度与容错率高——导出的数据文件可以作为文件缓存来使用, 而且爬取程序可以不过多理会爬取到的数据的正确性(数据类型, 是否为空等), 爬取程序就只管取与写。相对应的数据库程序则可以赢得更多的时间对数据进行校验与检查, 仅仅将有效且正确的数据批量地保存到目标数据库中。

由于爬取与数据库存储分属两个独立进程, 会带来以下缺点:

(1) 进程的运行时机难以把控——爬取程序的优先级必然高于数据库存储程序。也就是说, 数据库导入程序需要等待爬取程序完成后才能启动, 如果同时运行, 则会导致两个程序同时读写文件甚至删除文件而引发文件 I/O 异常。在爬取程序处于长期运行的场景下, 两个程序的运行时机很难把握。

(2) 数据的即时性不能保证。

是否就没有既可以保持数据库的写入速度，又能保证蜘蛛的稳定性和实时性的方案呢？方法总比问题多，我和我的团队经过长时间的实践与思考，最终找到了一种方案：采用非结构化数据库即 NoSQL 数据库作为爬取数据的后端数据存储方案。NoSQL 数据库比传统的 SQL 数据库在大数据生态中占有更多的优势。例如，非结构化存储、分布式存储和面向超大型数据的集群式存储等方面的特性。

知名的 NoSQL 数据库有很多，例如，Redis、Hodoop、MongoDB 和 Cassandra 等，具体要看实际运用时哪一种 NoSQL 数据库更合适了。

```
import redis

class ArticleRedisFeedExporter(object):

    @classmethod
    def from_crawler(cls, crawler):
        try:
            exporter = cls.from_settings(crawler.settings)
            crawler.signals.connect(exporter.spider_opened,
signals.spider_opened)
            crawler.signals.connect(exporter.spider_closed,
signals.spider_closed)
            exporter.host = crawler.settings['REDIS_HOST']
            exporter.port = crawler.settings['REDIS_PORT']
        except AttributeError:
            exporter = cls()

        exporter.crawler = crawler
        return exporter

    def spider_opened(self, spider):
        # 创建与 Redis 的连接
        pool = redis.ConnectionPool(host=self.host)
        r = redis.Redis(connection_pool=pool)
        self.pipe = r.pipeline(transaction=True)

    def process_item(self, item, spider):
        ## article = Article(**item)
        _id = 'article:%s' % item['pub_date']
```



```
self.pipe.set('%s:title' % _id, item['title'])
self.pipe.set('%s:desc' % _id, item['desc'])
```

```
def spider_closed(self, spider):
    # 批量写入数据库
    pipe.execute()
```

## 4.6.5 示例：基于阿里云的存储后端

使用 Feed 导出时，可以使用 URI（通过 FEED\_URI 设置）定义在哪里存储 Feed。Feed 导出支持由 URI 方案定义多个存储后端类型。

Scrapy 支持开箱即用的存储后端包括：

- 本地文件系统；
- FTP；
- S3（需要 botocore 或 boto）；
- 标准输出。

如果所需的外部库不可用，则某些存储后端可能无法使用。例如，S3 后端仅在安装了 botocore 或 boto 库时才可用（Scrapy 仅支持 boto 到 Python 2）。

### 存储URI参数

前文只讲述了如何配置输出文件的格式，但文件将保存在哪个位置呢？文件的命名规则又是如何指定的呢？此时我们就需要配置 FEED\_URI 参数了，下文简称为存储 URI。

存储 URI 只是一个字符串，但它可以支持通配符，可以包含在创建订阅源时被替换的参数。这些参数是：

- %(time)s——在创建订阅源时由时间戳替换；
- %(name)s——被蜘蛛名替换。

任何其他命名参数将替换为同名的 spider 属性。例如，在创建订阅源的那一刻，%(site\_id)s 将被 spider.site\_id 属性替换。

存储在 FTP 中，使用每个蜘蛛一个目录：

```
ftp://user:password@ftp.example.com/scraping/feeds/%(name)s/%(time)s.json
```

存储在 S3 中，使用每个蜘蛛一个目录：

```
s3://mybucket/scraping/feeds/%(name)s/%(time)s.json
```

### 存储后端

使用哪种存储后端，存储到哪个位置都只通过 `FEED_URI` 设定，Scrapy 会自动从指定的 URI 协议中分析采用哪一种后端作为存储。归纳起来有以下几种，如下表所示。

协 议	说 明
file://	本地文件系统
s3://	Amazon S3 存储（URI 中可带有用户名与密码参数）
ftp://	标准 FTP 的地址格式（URI 中可指定 FTP 用户名与密码）
stdout:	标准输出

#### ➤ 本地文件系统

订阅源存储在本地文件系统中。

示例 URI：

```
FEED_URI='file:///tmp/export.csv'
```

注意，对于本地文件系统存储，如果指定绝对路径，则可以省略该方案 `/tmp/export.csv`。这只适用于 UNIX 系统。

#### ➤ FTP

订阅源存储在 FTP 服务器中。

示例 URI：

```
FEED_URI='ftp://user:pass@ftp.example.com/path/to/export.csv'
```

#### ➤ S3

订阅源存储在 Amazon S3 上。

示例 URI：

```
s3://mybucket/path/to/export.csv
```

```
s3://aws_key:aws_secret@mybucket/path/to/export.csv
```



**注：**所需的外部库为 `botocore` 或 `boto`。

AWS 凭证可以作为 URI 中的用户/密码传递，也可以通过以下设置传递：

```
AWS_ACCESS_KEY_ID
AWS_SECRET_ACCESS_KEY
```

### ➤ 标准输出

`Feed` 被写入 `Scrapy` 进程的标准输出，即时打印到屏幕上。

示例 URI：

```
FEED_URI='stdout:'
```

### 自定义存储后端

`Scrapy` 的存储后端的设计思路是非常好的，它既实用又遵循了 `Scrapy` 的核心思想——可扩展。但 `Scrapy` 的标准文档中并没有说明如何去扩展出我们所需要的定制化的后端存储方式。此处就由外而内地分析一下如何为 `Scrapy` 扩展更多的存储方式。

首先，可以从后端存储的标准配置项 `FEED_STORAGES_BASE` 入手，配置代码如下所示。

```
FEED_STORAGES_BASE = {
    '': 'scrapy.extensions.feedexport.FileFeedStorage',
    'file': 'scrapy.extensions.feedexport.FileFeedStorage',
    'stdout': 'scrapy.extensions.feedexport.StdoutFeedStorage',
    's3': 'scrapy.extensions.feedexport.S3FeedStorage',
    'ftp': 'scrapy.extensions.feedexport.FTPFeedStorage',
}
```

`FEED_STORAGES_BASE` 就是一个字典，用于配对后端协议类型与 `Python` 处理类的关系，这个与上文详细讲述的内容是完全对应的。也就是说，我们要扩展出新的后端存储，只要将具体的后端存储类配置到这里就行了。

接下来，写一个可以支持阿里云对象存储的后端。先从配置入手，即使现在还没有写出任何类，但由于配置是 `Scrapy` 程序的入口，因此这里也是程序的起点：

```
FEED_URI = 'ali:///oss-cn-hangzhou.aliyuncs.com/%(name)s/%(time)s.json',
FEED_STORAGES_BASE={
    ... , # 与上文相同，省略
```

```
'ali': 'myproject.feedexport.OSSFeedStorage'
}
```

接下来就是实现 `OSSFeedStorage` 类了,而在此之前需要先了解 `Scrapy` 存储后端的类结构。存储后端都实现了 `IFeedStorage` 接口,该接口的定义如下:

```
class IFeedStorage(Interface):
    """所有的存储后端口都必须实现此接口"""

    def __init__(uri):
        """通过指定的 URI 参数初始化存储"""

    def open(spider):
        """为指定的蜘蛛打开存储。此方法必须返回一个类似文件的对象用作数据导出"""

    def store(file):
        """存储文件流"""
```

由于存储到远端网络上,所以这个存储过程通常来说都是异步的,因此 `Scrapy` 提供了另一个很有用的名为 `BlockingFeedStorage` 的类,以下是该类的实现代码:

```
@implementer(IFeedStorage)
class BlockingFeedStorage(object):

    def open(self, spider):
        return TemporaryFile(prefix='feed-')

    def store(self, file):
        return threads.deferToThread(self._store_in_thread, file)

    def _store_in_thread(self, file):
        raise NotImplementedError
```

`BlockingFeedStorage` 实现了一个异步线程,同时简化了 `IFeedStorage` 接口的实现,我们只需要实现 `__init__` 并初始化,然后在 `_store_in_thread` 中实现存储逻辑即可。

按照这个思路, `OSSFeedStorage` 的实现如下:

```
class OSSFeedStorage(BlockingFeedStorage):
```



```

def __init__(self, uri):
    from scrapy.conf import settings
    try:
        import oss2
    except ImportError:
        raise NotConfigured

    # self.connect_s3 = boto.connect_s3

    u = urlparse(uri)
    self.bucketname = u.hostname
    self.access_key = u.username or settings['OSS_ACCESS_KEY_ID']
    self.secret_key = u.password or settings['OSS_SECRET_ACCESS_KEY']
    self.keyname = u.path

def _store_in_thread(self, file):
    file.seek(0)
    auth = oss2.Auth(self.access_key, self.secret_key)
    # bucket = oss2.Bucket(auth, 'http://oss-cn-hangzhou.aliyuncs.com',
'bucket 名称') 阿里云的用法参考
    bucket = oss2.Bucket(auth, 'http://' + u.hostname, self.bucketname)
    bucket.put_object_from_file(file.name, file.name)

    # conn = self.connect_s3(self.access_key, self.secret_key)
    # bucket = conn.get_bucket(self.bucketname, validate=False)
    # key = bucket.new_key(self.keyname)
    # key.set_contents_from_file(file)
    # key.close()

```

由于访问阿里云 OSS 需要提供该存储的公钥与私钥（在阿里云的管理后台可以获得），因此需要在配置文件中加入这两个设置：

```

OSS_ACCESS_KEY_ID = 'OSS 上访问公钥'
OSS_SECRET_ACCESS_KEY = 'OSS 的访问私钥'

```

另外，运行以上代码之前需要先安装 oss2（由阿里云提供访问工具包）：

```
$ pip install oss2
```

## 设置

Scrapy 还提供了下表中的配置，用于控制文件的导出。

配置项	说明
FEED_FORMAT	要用于 Feed 的序列化格式
FEED_STORAGEES	项目支持的 Feed 存储后端的字典。键是 URI 方案，值是存储类的路径
FEED_EXPORTERS	使用的 Exporters 的字典，键为导出格式，值为导出类
FEED_STORE_EMPTY	是否导出空 Feed（即没有项目的 Feed）
FEED_EXPORT_ENCODING	用于导出文件内容的文字编码。如果设置为 None（默认），则不进行显式的声明，它将自动使用 UTF-8 进行编码
FEED_EXPORT_FIELDS	导出的字段的有序列表，如 FEED_EXPORT_FIELDS = ["foo", "bar", "baz"]
FEED_URI	导出文件的保存位置
AWS_ACCESS_KEY_ID	仅 S3 使用
AWS_SECRET_ACCESS_KEY	仅 S3 使用

以下为完整的配置示例：

```
FEED_URI = None
FEED_URI_PARAMS = None # a function to extend uri arguments
FEED_FORMAT = 'jsonlines'
FEED_STORE_EMPTY = False
FEED_EXPORT_FIELDS = None
FEED_STORAGEES = {}
FEED_STORAGEES_BASE = {
    '': 'scrapy.extensions.feedexport.FileFeedStorage',
    'file': 'scrapy.extensions.feedexport.FileFeedStorage',
    'stdout': 'scrapy.extensions.feedexport.StdoutFeedStorage',
    's3': 'scrapy.extensions.feedexport.S3FeedStorage',
    'ftp': 'scrapy.extensions.feedexport.FTPFeedStorage',
}
FEED_EXPORTERS = {}
FEED_EXPORTERS_BASE = {
    'json': 'scrapy.exporters.JsonItemExporter',
    'jsonlines': 'scrapy.exporters.JsonLinesItemExporter',
    'jl': 'scrapy.exporters.JsonLinesItemExporter',
    'csv': 'scrapy.exporters.CsvItemExporter',
```



```

    'xml': 'scrapy.exporters.XmlItemExporter',
    'marshal': 'scrapy.exporters.MarshalItemExporter',
    'pickle': 'scrapy.exporters.PickleItemExporter',
}

```

### 附: Python与SQL

数据库表是一个二维表, 包含多行多列。一个表的内容可以用 Python 的数据结构表示出来, 一个 list 表示多行, list 的每一个元素是 tuple, 表示一行记录。比如, 包含 id 和 name 的 user 表:

```

[
    ('1', 'Michael'),
    ('2', 'Bob'),
    ('3', 'Adam')
]

```

Python 的 DB-API 返回的数据结构就是像上面这样表示的。

但是用 tuple 表示一行很难看出表的结构。如果把一个 tuple 用 class 实例来表示, 则可以更容易地看出表的结构:

```

class User(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

[
    User('1', 'Michael'),
    User('2', 'Bob'),
    User('3', 'Adam')
]

```

这就是 ORM 技术, 把关系数据库的表结构映射到对象上。

虽然数据库多种多样, 但从对象关系映射 (Object Relational Mapping, 简称 ORM) 模式被广泛应用之后, 对于数据库的客户端开发影响其实并不大。当下的主流语言几乎都拥有自身的 ORM 框架模型, 即使开发人员完全不懂 SQL, 也能开发 SQL 数据库应用。这在早年间几乎是不可想象的一件事, ORM 在没有被广泛认同之时, 不懂 SQL 语言就是不懂数据库开发。

对象关系映射模式是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说, ORM 通过使用描述对象和数据库之间映射的元数据, 将程序中的对象自动持久化到关系数据库中。

Python 原生语言包中并没有附带 ORM 工具包, 但 Python 的完善得益于它强大的社区贡献。其中最知名且应用最广泛的就数 SQLAlchemy (<http://www.sqlalchemy.org/>) 了, 我们只需要在项目中输入以下指令就能轻易地安装它:

```
$ pip install sqlalchemy
```

或

```
$ easy_install sqlalchemy
```

### SQLAlchemy使用简介

由于本书篇幅有限, 而且数据库与 SQLAlchemy 的相关话题所涉及的范围极广, 难以用短短的几行字数将其囊括。考虑到一些没有接触过 SQLAlchemy 的读者, 在此会简单地以代码为例介绍 SQLAlchemy 的基本用法, 更具体的内容可以到 SQLAlchemy 的官网中仔细学习。对于已经熟悉 SQLAlchemy 读者可以直接跳过本段进入下一部分的内容。

由于 ORM 是一种模式, 也就是说, 只要是 ORM, 其使用方法都是大同小异的, 归纳起来有以下 4 步:

(1) 建模——建立与数据库对等的对象模型。

(2) 建立数据连接并产生数据上下文——通过连接字符串与指定数据库建立连接并取得可操作当前数据库的上下文对象。

(3) 操作数据——通过数据库上下文对象对数据进行增加、删除、修改、查询等常规操作。

(4) 提交更改——将变更后的数据内容提交并永久性写入数据库。

#### ➤ 第一步：建模

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, Text, create_engine
from sqlalchemy.orm import sessionmaker
```

# 创建对象的基类:

```
db = declarative_base()
```

```
class Article(db):
```



```
__tablename__ = 'articles'

id = Column(Integer, primary_key=True)
title = Column(String(255))
body = Column(Text)
```

以上代码完成 SQLAlchemy 的初始化和具体每个表的 class 定义。如果有多个表，则继续定义其他 class。例如，Category 类与 Article 建立一对多的数据关系：

```
class Category(db):
    __tablename__ = 'categories'

    id = Column(Integer, primary_key=True)
    name = Column(String(255))
    articles = relationship('Article')

class Article(db):
    __tablename__ = 'articles'

    id = Column(Integer, primary_key=True)
    title = Column(String(255))
    body = Column(Text)
    # 建立外键关系，即“多”方
    category_id = Column(Integer, ForeignKey('categories.id'))
```

### ➤ 第二步：建立连接

```
# 初始化数据库连接:
engine = create_engine('mysql+mysqlconnector://root:password@localhost:
3306/test')

# 创建 DBSession 类型:
DBSession = sessionmaker(bind=engine)

class School(Base):
    __tablename__ = 'school'
    id = ...
    name = ...
```

create\_engine() 用来初始化数据库连接。SQLAlchemy 用一个字符串表示连接信息：

```
'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'
```

只需要根据需要替换掉用户名、口令等信息即可。

### ➤ 第三步：操作数据对象

下面我们看看如何向数据库表中添加一行记录。

由于有了 ORM，向数据库表中添加一行记录可以视为添加一个 User 对象：

```
# 创建 session 对象：
session = DBSession()

# 创建新 User 对象：
new_category = Category(id='5', name='Bob')

# 添加到 session 中：
session.add(new_category)
```

可见，关键是获取 session，然后把对象添加到 session 中，最后提交并关闭。session 对象可视为当前数据库连接。

如何从数据库表中查询数据呢？有了 ORM，查询出来的可以不再是 tuple，而是 User 对象。SQLAlchemy 提供的查询接口如下：

```
# 创建 Session：
session = DBSession()

# 创建 Query 查询，filter 是 where 条件，最后调用 one() 返回唯一行，如果调用 all() 则返回所有行：
category = session.query(Category).filter(Category.id=='5').one()

# 打印类型和对象的 name 属性：
print 'type:', type(category)
print 'name:', category.name

# 关闭 Session：
session.close()
```

运行结果如下：



```
type: <class '__main__.Category'>
name: News
```

可见, ORM 就是将数据库表的行与相应的对象建立关联, 互相转换。

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联, 相应地, ORM 框架也可以提供两个对象之间的一对多、多对多等功能。

例如, 如果一个 User 拥有多个 Book, 就可以定义一对多关系:

```
class User(object):

    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    books = db.relationship('Book',
                             backref='user',
                             lazy='dynamic')

class Book(object):

    __tablename__ = 'books'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'))
```

#### ➤ 第四步: 提交更改

由于所有的数据变更内容是存储在 session 的跟踪对象内的, 只有在调用 commit() 方法后, 所有的更改才会被真正写入数据库, 因此在完成数据的操作后需要调用以下语句:

```
# 提交即保存到数据库中:
session.commit()

# 关闭 session:
session.close()
```

## 附：Scrapy内置Feed Exporters参考

### ➤ BaseItemExporter

```
class scrapy.contrib.exporter.BaseItemExporter(fields_to_export=None,
export_empty_fields=False, encoding='utf-8')
```

这是所有 Item Exporters 的（抽象）父类。它为所有（具体）Item Exporters 提供基本属性，如定义“export”什么 fields，是否“export”空 fields，是否进行编码。

可以在构造器中设置它们不同的属性值：fields\_to\_export、export\_empty\_fields、encoding。

#### **export\_item(item)**

输出给定 Item，此方法必须在子类中实现。

#### **serialize\_field(field, name, value)**

返回给定 field 的序列化值，可以覆盖此方法来控制序列化或输出指定的 field。

默认情况下，此方法寻找一个 serializer，在 item field 中声明并返回它的值。如果没有发现 serializer，则值不会改变，除非使用 unicode 值并编码到 str，编码可以在 encoding 属性中声明。

参数：

- field (Field 对象) —— 指定用于序列化的字段对象实例。
- name (str) —— 序列化字段的名称。
- value —— 用于序列化的值。

#### **start\_exporting()**

表示 exporting 过程的开始。一些 exporters 用于产生需要的头元素（例如，XmlItemExporter）。在实现 exporting item 前必须调用此方法。

#### **finish\_exporting()**

表示 exporting 过程的结束。一些 exporters 用于产生需要的尾元素（例如，XmlItemExporter）。在完成 exporting item 后必须调用此方法。

#### **fields\_to\_export**

列出“export”什么 fields 值，None 表示导出所有字段，默认值为 None。

一些 exporters（例如，CsvItemExporter）会按照定义 fields 属性中的次序依次输出。



**export\_empty\_fields**

是否在输出数据中包含为空的 item fields, 默认值是 False。一些 exporters (例如, CsvItemExporter) 会忽略此属性并输出所有字段。

**encoding**

encoding 属性将用于编码 Unicode 值 (仅用于序列化字符串), 其他值类型将不变地传递到指定的序列化库。

**➤ XmlItemExporter**

```
class scrapy.contrib.exporter.XmlItemExporter(file, item_element='item',
root_element='items', **kwargs)
```

以 XML 格式 “exports” Items 到指定的文件类。

参数:

- file——文件类;
- root\_element (str)——XML 根元素名;
- item\_element (str)——XML item 的元素名。

构造器额外的关键字参数将传给 BaseItemExporter 构造器。

一个典型的 exporter 实例:

```
<?xml version="1.0" encoding="utf-8"?>
<items>
  <item>
    <name>Color TV</name>
    <price>1200</price>
  </item>
  <item>
    <name>DVD player</name>
    <price>200</price>
  </item>
</items>
```

除了覆盖 serialize\_field() 方法, 多个值的 fields 会转化每个值到<value>元素。

例如, Item:

```
Item(name=['John', 'Doe'], age='23')
```

将被转化为:

```
<?xml version="1.0" encoding="utf-8"?>
<items>
  <item>
    <name>
      <value>John</value>
      <value>Doe</value>
    </name>
    <age>23</age>
  </item>
</items>
```

#### ➤ CsvItemExporter

```
class scrapy.contrib.exporter.CsvItemExporter(file, include_headers_line=
True, join_multivalued=', ', **kwargs)
```

输出 CSV 文件格式。如果添加 `fields_to_export` 属性,则它会按顺序定义 CSV 的列名。  
`export_empty_fields` 属性在此没有作用。

参数:

- `file`——文件类;
- `include_headers_line`——启用后 `exporter` 会输出第一行为列名,列名从 `BaseItemExporter.fields_to_export` 或第一个 `item fields` 中获取。
- `join_multivalued`——将用于连接多个值的 `fields`。此构造器额外的关键字参数将传给 `BaseItemExporter` 构造器,其余的将传给 `csv.writer` 构造器,以此来定制 `exporter`。

一个典型的 `exporter` 实例:

```
product,price
Color TV,1200
DVD player,200
```

#### ➤ PickleItemExporter

```
class scrapy.contrib.exporter.PickleItemExporter(file, protocol=0, **kwargs)
```



输出 pickle 文件格式。

参数:

- file——文件类;
- protocol——pickle 协议。

此构造器额外的关键字参数将传给 BaseItemExporter 构造器。

pickle 不是可读的格式, 这里不提供实例。

#### ➤ PprintItemExporter

```
class scrapy.contrib.exporter.PprintItemExporter(file, **kwargs)
```

输出整齐打印的文件格式。

参数:

- file——文件类。

此构造器额外的关键字参数将传给 BaseItemExporter 构造器。

一个典型的 exporter 实例:

```
{'name': 'Color TV', 'price': '1200'}  
{'name': 'DVD player', 'price': '200'}
```

此格式会根据行的长短进行调整。

#### ➤ JsonItemExporter

```
class scrapy.contrib.exporter.JsonItemExporter(file, **kwargs)
```

输出 JSON 文件格式, 所有对象将写进一个对象的列表。此构造器额外的关键字参数将传给 BaseItemExporter 构造器, 其余的将传给 JSONEncoder 构造器, 以此来定制 exporter。

参数:

- file——文件类。

一个典型的 exporter 实例:

```
[{"name": "Color TV", "price": "1200"},  
{"name": "DVD player", "price": "200"}]
```

**注意：**JSON 是一个简单而有弹性的格式，但对大量数据的扩展性不是很好，因为这里会将整个对象放入内存。如果要 JSON 既强大又简单，则可以考虑 `JsonLinesItemExporter`，或把输出对象分为多个块。

#### ➤ `JsonLinesItemExporter`

```
class scrapy.contrib.exporter.JsonLinesItemExporter(file, **kwargs)
```

输出 JSON 文件格式，每行写一个 JSON-encoded 项。此构造器额外的关键字参数将传给 `BaseItemExporter` 构造器，其余的将传给 `JSONEncoder` 构造器，以此来定制 exporter。

参数：

- `file`——文件类。

一个典型的 exporter 实例：

```
{"name": "Color TV", "price": "1200"}  
{"name": "DVD player", "price": "200"}
```

这个类能很好地处理大量数据。



# 5 chapter

## 第 5 章 高阶虫术

当真正理解中阶虫术中提及的相关技术并将它们应用到实战后，本章将是一个新的开始。在经过大量的理论学习与实践打磨之后，我们需要的是技术上的升华。

虫术的根本不单单是技术的应用，而是一种对数据采集系统化的设计思路。

虫术最难的并不是如何掌握某种技术从而实现极为炫酷的功能，虫子的任务永远都只是单一与纯正的——收集数据。

简言之，高阶虫术是对中级虫术的进一步深化，尤其在处理的细节上，正所谓“细节决定成败”，在这一阶段将会讲述以下内容：

- 如何开发具有高性能的爬虫；
- 让爬虫无坚不摧所向披靡；
- 从“单干”到“群战”；
- 可视化爬虫。

### 如何开发具有高性能的爬虫

在得到“同等数量爬取结果”的情况下，好的爬虫系统应该：

- (1) 蜘蛛出动的次数越少越好。
- (2) 蜘蛛每次运行的周期越短越好（内存损耗越小越好）。
- (3) 蜘蛛的生存度（不被封杀）越高越好。

(4) 蜘蛛的数量越多、分布越广、成本越低越好。

(5) 蜘蛛越隐秘越好。

### 让爬虫无坚不摧所向披靡

- 如何让虫看起来更像人？
- 如何部署超大规模的虫群？
- 如何让虫更聪明？
- 如何让虫更高效地运行？
- 如何让虫不被识别，变得更加隐蔽？
- 如何消化虫爬取的海量数据？

### 建立大规模的“爬虫大军”，从“单干”到“群战”。

正所谓人多力量大，爬虫的运行需要消耗大量的服务器资源，尤其对于大规模的持久性增量式爬网，服务器的负荷是非常大的。如果通过提升机器的性能来应对，则会增加爬虫系统的运行成本，而且成本有可能持续增加。分布式爬虫可以有效地将这种爬网负荷分散到网络中的各个爬虫节点上，充分利用网络资源之余还可以有效合理地利用服务器资源，降低运行成本。

### 可视化爬虫

当全部掌握虫术以后，爬虫项目的开发应该是非常容易的，是否还有更高效的办法呢？答案显然是肯定的，你曾想过将那些枯燥的工作转化为可视化的交互，把那些固定的规则编写变成简单的鼠标单击吗？

## 5.1 增量式爬网

高阶虫术应对的主要是持久的大规模增量式爬网。很多问题与挑战在数据量小、爬虫运行频次低的场景下都是隐性的，一旦数据量日益膨胀，很多问题就会暴露出来，甚至成为一项技术的挑战。

### 面临的挑战

增量式爬网面临什么样的挑战呢？只要先将问题回归到本源即爬虫系统的本质：发起请求→提取数据→存储结果，然后将这个过程每天重复一亿次或者更多，沿着这个思路推演一下，爬虫系统所要面临的挑战就显而易见了。

#### ➤ 性能消耗大

爬虫系统最大的消耗就是网络资源，也就是网速与流量。对于上亿级别的爬取，单线程爬



虫明显是不能完成任务的, 那么处理并发就是必不可少的了。也就是说, CPU 的消耗也是持久性的。在面对这样的数量级爬取任务时, 我们就需要使用各种手段来提升性能。

- (1) 通过路由规则预先推演和生成目标 URL, 而不是通过跟进链接 (follow) 来获取 URL。
- (2) 减少爬虫系统发出没有必要的、重复性的网络请求。
- (3) 采用分布式爬虫架构, 将爬网的负荷分散到网络各处。

#### ➤ 反爬机制的阻挠

一旦爬虫系统以大规模的并发方式“野蛮”地采集网页数据, 大多数情况下都会与反爬网机制正面遭遇, 最终落得被封禁的下场。如何绕过对方网站的反爬网机制, 让爬虫既能保持高效的采集性能, 又能隐匿在众多的网络访客之中, 成为了增量式爬网系统的挑战之一。

#### ➤ 数据量大

在面对海量的数据收集结果时, 数据的存储、缓存、校验、入库将成为爬虫系统最后的一块绊脚石。

本章将重点放在如何降低爬虫系统的性能消耗, 科学有效地提高爬虫系统运行的效能上。

### 5.1.1 推演路由

路由是一种或多种用于生成 URL 的规则, 一般来说, 在网站开发的初期就开始制定路由。这是所有现代后端服务开发框架都具备的一种功能, 甚至已经被引入一些流行的前端开发框架中。例如, Vue、Angular 和 React 等。

在初、中级阶段我们并不在意爬虫爬取的 URL 的数量, 从惯性思维上来说, 只要有像 CrawlSpider 这样的蜘蛛, 将页面内的链接一次性提取出来, 然后统统爬取一次不就行了吗? 诚然, 这是最简单、直接的办法, 却不是最好的做法。

在得到同等数量爬取结果的情况下, 蜘蛛出动的次数越少越好。

这是评估爬虫系统好坏的第一条标准。如果使用前文中 CrawlSpider 的扫荡性的爬取方法, 显然会出现大量无用的请求, 除了爬取效率低下, 还容易触发被爬取目标网站的反爬虫机制。在某些网站上经常会利用 CrawlSpider 从入口页面获取进入链接的特性预埋各种爬虫“地雷”, 蜘蛛一旦“触雷”就会被直接屏蔽。

有用请求与无用请求混杂是最容易触发频繁异常请求这种常规的反爬规则的做法, 即使通过 Scrapy 的配置降低爬虫的并发数量, 但还是会影响系统的整体运行效率, 这是因为无效请求所导致的。

与其通过蜘蛛对响应内容分析来获取下一级别的爬网地路径，不如从一开始就推算对方网站的 URL 的生成规则，再通过规则直接生成要爬取的网页地址清单，这样既减少了被对方反爬系统发现的风险，又大大提高了爬虫系统本身的性能，同时节约了大量的时间（在面对可能上亿条数据的获取需求时，每个请求增加 1ms，那等待时间就可以想象了）。

推算的方法其实非常简单，现在流行的 Web 开发框架都得先设计路由规则（Routes）再对相应的网页进行设计。由于各种语言框架都希望开发人员的学习曲线变得平缓一些，因此大多数路由设计都是约定俗成的，而且好的 URL 都是人机可读且便于记忆或者方便推算的。

例如，常见的博客路由规则：

```
http://domain.com/blogs/2017-11-01/this-is-a-blog.html
```

可以得出这样的结论（<>内代表规则）：

```
http://domain.com/blogs/<日期>/<标题>.html
```

更典型的是以整数为增量计算型的：

```
http://domain.com/products/items/1
```

规则就应该是：

```
http://domain.com/products/items/<产品 ID>
```

在 Scrapy 中可以通过控制蜘蛛的 `start_urls` 来生成 URL，以上文中的产品 URL 为例：

```
class MySpider(Spider):

    def gen_urls(starts, ends):
        for i in ranges(starts, ends):
            yield "http://domain.com/products/items/%d" % i

    start_urls = gen_urls(1, 2000)

    def parse(self, response):
        # 省略 ...
```



上述代码中 `gen_urls` 有 `starts` 和 `ends` 两个参数, 用于界定生成的产品 ID 的范围。实际情况下, 我们并不会知道对方网站到底会有多少个产品, 为了可以将生成范围覆盖全部的产品 ID, 可以将 `ends` 尽量设置得大一些。这样就有可能生成一些对方网站上完全没有的 URL, 一旦将这些 URL 请求发送到对方服务器, 必然会产生大量的 404 (没有找到 URL) 错误, 这样会很容易引起对方网站的警觉。所以我们应该检测这个最大范围, 如果连续产生 3 个以上的 404 错误, 则可以推算产品 ID 已经到达上限, 后面的 URL 已经无效了, 可以让系统停止派出蜘蛛。

此时就需要使用 Scrapy 提供的两个非常有用的中间件来快速解决上面的问题。

首先, 对于 4xx 的错误可以选择跳过, 使用 `HttpErrorMiddleware` 来过滤 4xx 错误, 不需要进行额外的编码, 只在 `settings.py` 中增加相关的配置即可。在默认情况下, 这个中间件是被 Scrapy 设置为启用状态的, 只要修改该中间件的配置项即可, 具体做法如下:

```
HTTPERROR_ALLOWED_CODES = [200, 201, 304]
                                # 只处理正常响应和重定向请求, 其他请求一律过滤
HTTPERROR_ALLOW_ALL = False    # 设置为 True, 会忽略所有的 HTTP 错误
```

除了要过滤 4xx 类的大量无意义的 URL, 在发起大量的请求期间, 某些请求可能由于网络通信上的不可预知问题导致请求失败, 我们还应该启用 `RetryMiddleware` 下载器中间件, 该中间件将重试可能由于临时问题导致的请求失败。例如, 连接超时或者 HTTP 500 错误导致失败的页面。这个中间件由以下三个配置参数控制:

```
RETRY_ENABLED = True           # 启用重试
RETRY_TIMES = 3                # 重试的次数
RETRY_HTTP_CODES = [500]       # 指定需要重试的响应状态码
```

## 5.1.2 时机的重要性

相信大多数开发人员开发爬虫系统的主要目的是收集数据, 拖垮对方网站是一种不道德的行为, 所以我们得注意爬取的时间与速度。前文也提到由于 Scrapy 是并发式处理的, 所以每一次发动爬虫它们都会以默认的并发数同时向目标进发 (Scrapy 是每秒 16 个并发, 这是非常高的速度), 对于一些性能不高的网站, 这种爬取速度在某些时候 (例如, 在每天访问量最大的时间) 等同于攻击, 直接将服务器的带宽与容量耗光而将其 “拖死”。

优秀的爬虫系统应该具有潜行的能力, 应该在不知不觉中获取数据。尤其是要进行长期运行的增量式爬虫系统, 将爬虫装扮得像一个真人混在正常的访问量中, 不要做出非人的行为; 或者选择在对方服务器闲置的时间 (深夜) 才出来运行; 至少不要引起对方系统的 “关注”。

在对的时间做对的事——不但可以用于做人，同样也可以用于设计好的爬虫。

### 5.1.3 去重处理

去重处理可以避免将重复性的数据保存到数据库中以造成大量的冗余性数据。

不要在获得蜘蛛爬网结果后进行内容过滤，这样做只不过是避免后端数据库出现重复数据。去重处理对于一次性爬取是有效的，但对于增量式爬网则恰恰相反。对于持续性长的增量式爬网，应该进行“前置过滤”，这样可以有效地减少蜘蛛出动的次数。

在发出请求之前检查蜘蛛是否曾爬取过该 URL，如果已爬取过，则让蜘蛛直接跳过该请求以避免重复出动。除了重复的 URL 指纹，还应该加上 404 与 500 错误的 URL 过滤，因为即使目标网站上没有反爬网机制，但绝大多数的 Web 服务器程序都会有对 404 与 500 错误的记录。过多的 404 与 500 很容易暴露蜘蛛的痕迹，因此加入对异常 URL 的筛选是非常有必要的。

Scrapy 提供了一个很好的请求指纹过滤器（Request Fingerprint duplicates filter）`scrapy.dupefilters.RFPDupeFilter`，当它被启用后，会自动记录所有成功返回响应的请求的 URL 并将其以文件（`requests.seen`）方式保存在项目目录中。请求指纹过滤器的原理是为每个 URL 生成一个指纹并记录下来，一旦当前请求的 URL 在指纹库中有记录，就自动跳过该请求。

默认情况下这个过滤器是被自动启用的。当然也可以根据自身的需求编写自定义的过滤器，继承 `scrapy.dupefilters.BaseDupeFilter` 来开发自定义的过滤器。

```
class BaseDupeFilter(object):

    @classmethod
    def from_settings(cls, settings):
        return cls()

    def request_seen(self, request):
        """
        返回一个布尔值。当请求已重复时返回真，否则返回假。
        """
        return False

    def open(self):
        """
```



当过滤器被打开时执行

```
"""
```

```
    pass
```

```
def close(self, reason):
```

```
"""
```

当过滤器被关闭时执行

```
"""
```

```
    pass
```

```
def log(self, request, spider):
```

```
"""
```

记录请求已被过滤

```
"""
```

```
    Pass
```

由于 `scrapy.dupefilters.RFPDuperFilter` 采用文件方式保存指纹库, 对于增量爬取且只用于短期运行的项目还能应对。一旦遇到爬取量巨大的场景时, 这个过滤器就显得不太适用了, 因为指纹库文件会变得越来越大, 过滤器在启动时会一次性将指纹库中所有的 URL 读入, 导致消耗大量内存。

可以用 Scrapy 提供的 `request_fingerprint` 函数为请求生成指纹, 然后将指纹写入 Redis 中, 实现代码如下:

```
from redis import StrictRedis
from scrapy.utils.request import request_fingerprint
import logging

class RedisDupeFilter(object):

    def __init__(self):
        self.redis = StrictRedis(port=REDIS_PORT, db=REDIS_DUP_DB)
        self.logger = logging.getLogger(__name__)

    @classmethod
    def from_settings(cls, settings):
        redis_port = settings.getint('REDIS_PORT')
        redis_db = settings.get('REDIS_DUP_DB')
```

```

        return cls(redis_port, redis_db)

    def request_seen(self, request):
        fp = self.request_fingerprint(request)
        key = 'UrlFingerprints'
        if self.redis.sismember(key, fp) is None:
            self.redis.sadd(key, fp)
            return False
        return True

    def log(self, request, spider):
        msg = ("已过滤的重复请求: %(request)s")
        self.logger.debug(msg, {'request': request}, extra={'spider': spider})
        spider.crawler.stats.inc_value('dupefilter/filtered', spider=spider)

```

在配置文件中启用这个过滤器：

```
DUPEFILTER_CLASS = 'my_crawler.dupefilters.RedisDupeFilter'
```

至此我们已接触了两种去重过滤器，接下来将介绍一个更强大的布隆过滤器。在此之前有必要对去重过滤器的应用先做一个简单的小结，归纳它们适用的场合才能更好地发挥它们的作用。

- 当数据量不大时（大约在 200MB 内），可以直接在内存中进行去重处理（例如，可以使用 `set()` 进行去重），而更省事又能对去重状态进行持久化的办法就是采用 `scrapy.dupefilters.RFPDupeFilter`；
- 当数据量在 5GB 以内时，建议采用上文中的 `RedisDupeFilter` 进行去重，当然这要求服务器的内存必须大于 5GB，否则 Redis 可能会将机器的内存耗光；
- 当数据量达到 10~100GB 级别时，由于内存有限，就必须用“位”来去重，才能够满足需求。而布隆过滤器就是将去重对象映射到几个内存“位”，通过几个位的 0/1 值来判断一个对象是否已经存在，以应对海量级的请求数据的重复性校验。

#### 5.1.4 布隆过滤器

布隆过滤器（Bloom Filter）是由 Burton Howard Bloom 于 1970 年提出的，它是一种 `space efficient` 的概率型数据结构，用于判断一个元素是否在集合中。在垃圾邮件过滤的黑白名单方法、爬虫（Crawler）的网址判重模块等场景中经常被用到。哈希表也能用于判断元素是否在集合中，



但是布隆过滤器只需要哈希表的 1/8 或 1/4 的空间复杂度就能完成同样的任务。布隆过滤器可以插入元素,但不可以删除已有元素。元素越多, false positive rate(误报率)越大,但是 false negative(漏报)是不可能的。

在爬虫中使用布隆过滤器可以实现高效去重。布隆过滤器可以用于快速检索一个元素是否在集合中。布隆过滤器实际上是一个很长的二进制向量和一系列随机映射函数(Hash 函数)。而一般判断一个元素是否在集合中的做法是:用需要判断的元素和集合中的元素进行比较,大部分数据结构如链表、树,都是这么实现的。

## 优点

相比于其他数据结构,布隆过滤器在空间和时间方面都有巨大的优势。布隆过滤器存储空间和插入/查询时间都是常数  $O(k)$ 。另外,散列函数相互之间没有关系,方便由硬件并行实现。布隆过滤器不需要存储元素本身,在某些对保密要求非常严格的场合中非常有优势。布隆过滤器可以表示全集,其他任何数据结构都不能;  $k$  和  $m$  相同,使用同一组散列函数的两个布隆过滤器的交并来源请求运算可以使用位操作进行。

### ➤ 确定性

当使用相同大小和数量的哈希函数时,某个元素通过布隆过滤器得到的是正反馈还是负反馈的结果是确定的。对于某个元素  $x$ ,如果它现在可能存在,则五分钟之后、一小时之后、一天之后、甚至一周之后的状态都是可能存在的。当我得知这一特性时有一点惊讶。因为布隆过滤器是概率性的,其结果显然应该存在某种随机因素,难道不是吗?确实不是。它的概率性体现在我们无法判断究竟哪些元素的状态是可能存在的。

换句话说,过滤器一旦做出可能存在的结论后,结论就不会发生变化。

### ➤ 布隆过滤器的容量

布隆过滤器需要事先知道要插入元素的个数。如果并不知道或者很难估计元素的个数,则情况就不太妙。也可以随机指定一个很大的容量,但这样会浪费许多存储空间,存储空间是我们试图优化的首要任务,也是选择使用布隆过滤器的原因之一。一种解决方案是创建一个能够动态适应数据量的布隆过滤器,但是在某些应用场景下这个方案无效。有一种可扩展布隆过滤器,它能够调整容量来适应不同数量的元素,能够弥补一部分短板。

### ➤ 空间效率

如果想在集合中存储一系列的元素,则有很多种不同的做法。可以把数据存储在 HashMap 中,随后在 HashMap 中检索元素是否存在,HashMap 插入和查询的效率都非常高。但是,由于 HashMap 直接存储内容,所以空间利用率并不高。

如果希望提高空间利用率,则可以在元素插入集合之前做一次哈希变换。还有其他方法吗?

我们可以用位数组来存储元素的哈希值，也允许在位数组中存在哈希冲突。这正是布隆过滤器的工作原理，它们就是基于允许哈希冲突的位数组，可能会造成一些误报。在布隆过滤器的设计阶段就允许哈希冲突的存在，否则空间使用就不够紧凑了。

### 缺点

布隆过滤器的缺点和优点一样明显。误算率是其中之一。随着存入的元素数量增加，误算率随之增加。如果元素数量太少，则使用散列表。另外，一般情况下不能从布隆过滤器中删除元素。我们很容易想到把位数组变成整数数组，每插入一个元素相应的计数器加 1，这样删除元素时将计数器减掉就可以了。然而要保证安全地删除元素并非如此简单。首先我们必须保证删除的元素的确在布隆过滤器中。这一点单凭过滤器是无法保证的。另外计数器回绕也会造成问题。在降低误算率方面有不少方法，出现了很多布隆过滤器的变种。

#### ➤ 误报

布隆过滤器能够“拍着胸脯”说某个元素“肯定不存在”，但是对于一些元素它们会说“可能存在”。针对不同的应用场景，这有可能会是一个巨大的缺陷，或是无关紧要的问题。如果在检索元素是否存在时不介意引入误报情况，那么就应当考虑用布隆过滤器。

另外，如果随意地减小了误报比例，哈希函数的数量就要相应地增加，插入和查询时的延时也会相应地增加。本节的另一个要点是，如果哈希函数是相互独立的，并且输入元素在空间中均匀地分布，那么理论上真实误报率就不会超过理论值。否则，由于哈希函数的相关性和更频繁的哈希冲突，布隆过滤器的真实误报比例会高于理论值。

#### ➤ 布隆过滤器的构造和检索

在使用布隆过滤器时，我们不仅要接受少量的误报率，还要接受速度方面的额外时间开销。相比于 HashMap，对元素做哈希映射和构建布隆过滤器时必然存在一些额外的时间开销。

#### ➤ 无法返回元素本身

布隆过滤器并不会保存插入元素的内容，只能检索某个元素是否存在。因为存在哈希函数和哈希冲突，我们无法得到完整的元素列表。这是它相对于其他数据结构的显著优势，空间的使用率也造成了这块短板。

#### ➤ 删除某个元素

想从布隆过滤器中删除某个元素可不是一件容易的事情，我们无法撤回某次插入操作，因为不同项目的哈希结果可以被索引在同一位置。如果想撤销插入，则只能记录每个索引位置被置位的次数，或是重新创建一次。两种方法都有额外的开销。基于不同的应用场景，若要删除一些元素，则我们更倾向于重建布隆过滤器。



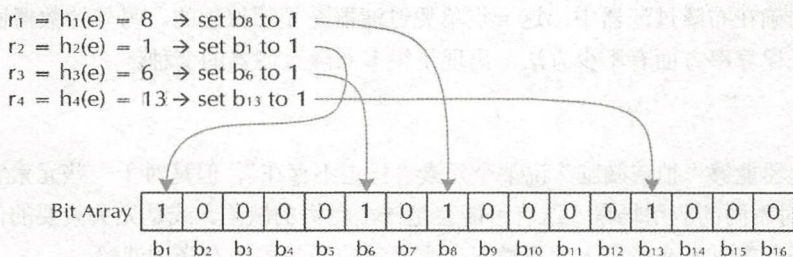
## 布隆过滤器的算法

创建一个  $m$  位 BitSet, 先将所有位初始化为 0, 然后选择  $k$  个不同的哈希函数。第  $i$  个哈希函数对字符串  $str$  哈希的结果记为  $h(i, str)$ , 且  $h(i, str)$  的范围是 0 到  $m-1$ 。

### ➤ 第一步: 加入字符串

下面是每个字符串处理的过程, 首先是将字符串  $str$  “记录” 到 BitSet 中:

对于字符串  $str$ , 分别计算  $h(1, str)$ 、 $h(2, str)$  …… $h(k, str)$ 。然后将 BitSet 的第  $h(1, str)$ 、 $h(2, str)$  …… $h(k, str)$  位设为 1。



### ➤ 第二步: 检查字符串是否存在

下面是检查字符串  $str$  是否被 BitSet 记录过的过程:

对于字符串  $str$ , 分别计算  $h(1, str)$ 、 $h(2, str)$  …… $h(k, str)$ 。然后检查 BitSet 的第  $h(1, str)$ 、 $h(2, str)$  …… $h(k, str)$  位是否为 1, 若其中任何一位不为 1, 则可以判定  $str$  一定没有被记录过。若全部位都是 1, 则“认为”字符串  $str$  存在。若一个字符串对应的 Bit 不全为 1, 则可以肯定该字符串一定没有被 BloomFilter 记录过 (这是显然的, 因为字符串被记录过, 其对应的二进制位肯定全部被设为 1 了)。若一个字符串对应的 Bit 全为 1 (实际上是不为 100% 的), 则肯定该字符串被 BloomFilter 记录过 (因为有可能该字符串的所有位都刚好被其他字符串所对应)。这种将该字符串划分错的情况, 称为“假阳性” (False Positive)。

## 布隆过滤器的参数选择

### ➤ 哈希函数的选择

哈希函数的选择对性能的影响应该是很大的, 一个好的哈希函数要能近似等概率地将字符串映射到各个 Bit。选择  $k$  个不同的哈希函数比较麻烦, 一种简单的方法是选择一个哈希函数, 然后送入  $k$  个不同的参数。

### ➤ $m$ 、 $n$ 、 $k$ 如何取值

- 可能把不属于这个集合的元素误认为属于这个集合 (假阳性, False Positive)。

- 不会把属于这个集合的元素误认为不属于这个集合（假阴性，False Negative）。

哈希函数的个数  $k$  取 10，位数组大小  $m$  设为字符串个数  $n$  的 20 倍时，假阳性发生的概率是 0.0000889，即 10 万次的判断中，会存在 9 次误判，对于一天 1 亿次的查询，误判的次数为 9000 次。

哈希函数的个数  $k$ 、位数组大小  $m$ 、加入的字符串数量  $n$  的关系可以参考下表。

m/n	k	k=17	k=18	k=19	k=20	k=21	k=22	k=23	k=24
22	15.2	2.67e-05							
23	15.9	1.61e-05							
24	16.6	9.84e-06	1e-05						
25	17.3	6.08e-06	6.11e-06	6.27e-06					
26	18	3.81e-06	3.76e-06	3.8e-06	3.92e-06				
27	18.7	2.41e-06	2.34e-06	2.33e-06	2.37e-06				
28	19.4	1.54e-06	1.47e-06	1.44e-06	1.44e-06	1.48e-06			
29	20.1	9.96e-07	9.35e-07	9.01e-07	8.89e-07	8.96e-07	9.21e-07		
30	20.8	6.5e-07	6e-07	5.69e-07	5.54e-07	5.5e-07	5.58e-07		
31	21.5	4.29e-07	3.89e-07	3.63e-07	3.48e-07	3.41e-07	3.41e-07	3.48e-07	
32	22.2	2.85e-07	2.55e-07	2.34e-07	2.21e-07	2.13e-07	2.1e-07	2.12e-07	2.17e-07

上表引用自 <http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html>。

### 布隆过滤器的Python实现

了解实现原理后，我们可以用“已有的轮子”来将布隆过滤器接入 Scrapy 中。pybloomfiltermmap (<https://github.com/axiak/pybloomfiltermmap>) 是 Python 世界中比较有名的布隆过滤器，按以下方式在命令行安装：

```
$ pip install pybloomfiltermmap
```

pybloomfiltermmap 的使用非常简单，它提供了一个 BloomFilter 类，在实例化时只需要输入布隆过滤器的大小与误判率与缓存文件名即可。先来看一个基本的使用示例，具体代码如下：

```
>>> fruit = pybloomfilter.BloomFilter(100000, 0.1, '/tmp/words.bloom')
>>> fruit.update(('apple', 'pear', 'orange', 'apple'))
>>> len(fruit)
3
>>> 'mike' in fruit
```



```
False
>>> 'apple' in fruit
True
```

update 方法是将需要判断的字符串加入布隆过滤器。

接下来就可以将其集成至 Scrapy 中, 编写一个 BloomDupeFilter, 使 Scrapy 能支持布隆去重, 具体代码如下所示。

```
from pybloomfilter import BloomFilter
from scrapy.utils.request import request_fingerprint

class BloomDupeFilter(object):

    def __init__(self):
        self.bloomfilter = BloomFilter(100000, 0.1, 'request_seen.bloom')

    def request_seen(self, request):
        fp = request_fingerprint(request)
        if fp in self.bloomfilter:
            return True
        else:
            self.bloomfilter.add(fp)
            return False

    def log(self, request, spider):
        msg = ("已过滤的重复请求: %(request)s")
        self.logger.debug(msg, {'request': request}, extra={'spider': spider})
        spider.crawler.stats.inc_value('dupefilter/filtered', spider=spider)
```

BloomDupeFilter 与 RFPDupeFilter 相比运行速度是有所提升的, 但是由于其持久化方式仍然采用文件形式, 加入文件时会将所有数据一次性地加载到系统的内存中, 一旦文件体积变得越来越大, 仍然无法逃脱“晒爆”内存的窘境。

### 5.1.5 基于Redis的布隆过滤器

要摆脱超大的持久化文件撑爆内存的问题, 最佳的解决办法还是将布隆过滤器持久化数据保存到数据库中。如果完全理解了布隆过滤器的算法与实现思路, 则一定会发现 Redis 可以作

为布隆过滤器的数据载体,Redis 和布隆过滤器简直就是天生一对!Redis 原生就有 BitSet 类型,非常容易操控。

首先需要实现一个 HashMap:

```
class HashMap(object):

    def __init__(self, m, seed):
        self.m = m
        self.seed = seed

    def hash(self, value):
        """
        哈希算法
        :param value: Value
        :return: Hash Value
        """
        ret = 0
        for i in range(len(value)):
            ret += self.seed * ret + ord(value[i])
        return (self.m - 1) & ret
```

然后是基于 Redis 实现的布隆过滤器:

```
BLOOMFILTER_HASH_NUMBER = 6
BLOOMFILTER_BIT = 30

class BloomFilter(object):

    def __init__(self, server, key='bloomfilters', bit=BLOOMFILTER_BIT,
hash_number=BLOOMFILTER_HASH_NUMBER):
        """
        构造 BloomFilter
        :param server: Redis 服务器地址
        :param key: 布隆过滤器在 Redis 中使用的键名
        :param bit: m = 2 ^ bit - 指定内存空间
        :param hash_number: 进行 Hash 的数量
        """
```



```

        # 默认  $1 \ll 30 = 10,7374,1824 = 2^{30} = 128\text{MB}$ , 最大过滤的请求指纹数为:
        2^30/hash_number = 1,7895,6970
        self.m = 1 << bit
        self.seeds = range(hash_number)
        self.server = server
        self.key = key
        self.maps = [HashMap(self.m, seed) for seed in self.seeds]

    def exists(self, value):
        """
        判断数据是否存于哈希表中
        :param value:
        :return:
        """
        if not value:
            return False
        exist = True
        for map in self.maps:
            offset = map.hash(value)
            exist = exist & self.server.getbit(self.key, offset)
        return exist

    def insert(self, value):
        """
        将数据插入布隆过滤器的键值存储空间内
        :param value:
        :return:
        """
        for f in self.maps:
            offset = f.hash(value)
            self.server.setbit(self.key, offset, 1)

```

最后接入 Scrapy 的过滤器框架:

```

from redis import StrictRedis
from redis_bloomfilter import BloomFilter
from scrapy.utils.request import request_fingerprint
import logging

```

```

class RedisBloomDupeFilter(object):

    def __init__(self, key, bit, hash_number):
        self.redis = StrictRedis(port=REDIS_PORT, db=REDIS_DUP_DB)
        self.key = key
        self.bit = bit
        self.hash_number = hash_number
        self.bf = BloomFilter(self.redis, key, bit, hash_number)
        self.logger = logging.getLogger(__name__)

    @classmethod
    def from_settings(cls, settings):
        redis_port = settings.getint('REDIS_PORT')
        redis_db = settings.get('REDIS_DUP_DB')
        bit = settings.getint('BLOOMFILTER_BIT', BLOOMFILTER_BIT)
        hash_number = settings.getint('BLOOMFILTER_HASH_NUMBER', BLOOMFILTER_
HASH_NUMBER)

        return cls(redis_port, redis_db)

    def request_seen(self, request):
        fp = request_fingerprint(request)
        if self.bf.exists(fp):
            return True

        self.bf.insert(fp)
        return False

    def log(self, request, spider):
        msg = ("已过滤的重复请求: %(request)s")
        self.logger.debug(msg, {'request': request}, extra={'spider': spider})
        spider.crawler.stats.inc_value('dupefilter/filtered',
spider=spider)

```

布隆过滤器的配置:



```
REDIS_DUP_DB = 0
BLOOMFILTER_HASH_NUMBER = 6
BLOOMFILTER_BIT = 30
```

### 小结

基于 Redis 的 Bloomfilter 去重，既用上了 Bloomfilter 的海量去重能力，又用上了 Redis 的可持久化能力，基于 Redis 也方便分布式机器的去重。在使用的过程中，要估算好待去重的数据量，根据上面的表，适当地调整 seed 的数量和 blockNum 数量（seed 越少去重速度越快，但漏失率越大）。

## 5.2 突破封印

蜘蛛在互联网上除了是数据的收集器，也可能是商业间谍、大规模流量作弊器、甚至是大规模攻击性武器。究其本质，爬虫就是互联网中带有攻击性的武器，这种特质是无法被磨灭的。至于它的好与坏与其自身毫无关系，而只在于用它的人。

当你在互联网上采用虫术爬取竞争对手的信息之时，有可能你的网站就会受到同样的待遇。了解对方如何发出攻击，就知道如何做出适当的防御。换句话说，爬网与反爬网永远是一对相互博弈又相互依存的话题。

“互联网不是战场”，但在大数据时代，数据变得越来越有价值，尤其是涉及各种商业活动的的数据。没有哪个商家在毫无利益的情况下愿意无条件地共享他们的数据，所以这只不过是一种互联网的“乌托邦”而已。爬虫与反爬虫的博弈甚至是战争，自爬虫被“生”下来那天开始就不曾停歇。

是的，从现在开始我们就得带着一种博弈思维来思考爬虫系统。爬虫世界并不存在任何文明与野蛮之别，而只有爬与反爬之分。

### 有礼貌地爬网

即使我们很清楚爬虫的世界就是一个“战场”，但战场也有战场的规矩。所有网站的反爬网机制都是一种防御机制，它们都执行唯一的准则：“虫不犯我，我不犯虫，虫若犯我，我必杀之”。要做到“不战而屈人之兵”，就得尊崇最基本的“道德”与“礼仪”。

爬虫的“道德标准”到底是什么呢？以下是我从实际经验中总结出来的几条：

- 学会尊重 robots.txt 文件的规则；
- 不能产生导致目标网站性能明显下降的行为；
- 采集数据后应该明确地标识内容的出处；



- 对方不愿意给你看的内容就“尽量”放下好奇心。

#### ➤ robots.txt

通俗地讲就是网站会通过 robots.txt 协议来自主控制是否愿意被搜索引擎收录，或者指定搜索引擎只收录指定的内容。而搜索引擎会按照每个网站赋予自己的权限来进行抓取。这就好比一个正常的人去别人家里，需要先敲门，得到许可后才能进入别人家。除非有主人的进一步许可和邀请，否则你不能擅自进入内室，或者在别人家里四处溜达。当然，强盗或者小偷例外。

这个协议也不是一个规范，而只是一个约定。有些搜索引擎会遵守这一约定，而其他的则不然。对搜索引擎如此，同样对于其他出于数据采集目的蜘蛛的作用也是如此。至于对方是否会遵守此文件内的约定，就得看蜘蛛方是否有“道德”了，所以说这只是一份“君子协定”。

robots 的基本用法：

- User-agent——这里代表所有的搜索引擎种类，\*是一个通配符。
- Disallow——虚目录。这里的定义是禁止爬寻指定目录下面的目录（可以带有多种通配符）。
- Allow——虚目录。这里定义允许爬寻指定目录下面的目录。
- Sitemap——网站地图。告诉爬虫这个页面是网站地图。
- Crawl-delay——对抓取程序设定一个较低的抓取请求频率。可以加入 Crawl-delay:xx 指示，其中“XX”是指在 Crawler 程序两次进入站点时，以秒为单位的最低延时。
- Visit-time——只有在 visit-time 指定的时间段中，robot 才可以访问指定的 URL，否则不可访问。
- Request-rate——用来限制 URL 的读取频率。

#### ➤ 如何运用Scrapy有礼貌地爬网

Scrapy 框架本身就提供了大量的中间件对爬网行为进行规范，在一定程度上可以避免那些“粗鲁”的爬网行为。即使如此，由于 Scrapy 是一个可高度定制与配置的框架，对于某些选项，我们在实际运行时还是需要加以检查和调整以达到实际的运行要求。

在 Scrapy 1.1+及以后的版本中，默认情况下一旦发现目标网站上有 robots.txt 协议文件，就会严格遵守该协议的内容。之前的版本可以在 settings.py 文件中加入以下选项以启用：

```
ROBOTSTXT_OBEY = True
```

之后，如果爬取对方禁止的内容时，则 Scrapy 会抛出如下调试信息：





```
2016-08-19 16:12:56 [scrapy] DEBUG: Forbidden by robots.txt: <GET
http://website.com/login>
```

### ➤ 延迟下载

Scrapy 的蜘蛛的工作效率非常高。它们可以同时处理多个并发请求并充分利用带宽和计算能力。但这种高效在很多情况下并不是好事,因为它很容易造成目标网站的性能下降,或者由于请求过于频繁、数据下载量过大而被对方网站发觉甚至直接被禁止访问。

要避免出现过频繁的访问请求,就需要在实际部署爬虫系统时设置 `DOWNLOAD_DELAY`。Scrapy 会在连续请求到相同域之间引入从  $0.5 * \text{DOWNLOAD\_DELAY}$  到  $1.5 * \text{DOWNLOAD\_DELAY}$  秒的随机延迟。如果想设置明确的 `DOWNLOAD_DELAY`,则必须禁用 `RANDOMIZE_DOWNLOAD_DELAY` 选项。

默认情况下 `DOWNLOAD_DELAY` 被设置为 0。如果要将每个请求之间的间隔设置为 5 秒,则可以按以下方式设置:

```
DOWNLOAD_DELAY = 5.0
```

如果项目中正在同时运行多个不同的蜘蛛,且蜘蛛之间的请求间隔又不尽相同时,则可以直接在蜘蛛中设置其 `download_delay` 属性以对配置进行单独的覆盖。

```
class MySpider(scrapy.Spider):
    name = 'myspider'
    download_delay = 5.0
    ...
```

### ➤ 调整并发请求数

可以调整每个域的并发请求数,使得蜘蛛变得更有“礼貌”。默认情况下,Scrapy 将最多同时向任何给定的域发送 8 个请求,但可以通过更新 `CONCURRENT_REQUESTS_PER_DOMAIN` 设置来更改此值。

**注意:** `CONCURRENT_REQUESTS` 用于设定 Scrapy 的下载器在同一时间的并发请求数。调整此设置对于服务器性能/带宽来说比在同一时间抓取多个域时的目标更高。

### ➤ AutoThrottle

网站可以处理的请求数量差别很大。如果对每个爬取的网站进行手动调整,则可能会让你抓狂。Scrapy 提供了一个名为 `AutoThrottle` 的扩展来解决这个问题。`AutoThrottle` 根据当



前 Web 服务器负载自动调整请求之间的延迟。它首先计算一个请求的延迟，然后调整同一个域的请求之间的延迟，使得不超过 `AUTOTHROTTLE_TARGET_CONCURRENCY` 的请求将同时激活。它还确保请求在给定的时间范围内均匀分布。

```
AUTOTHROTTLE_ENABLED = True
```

#### ➤ 在开发期间打开 HTTP Cache 功能

开发网络爬虫项目是一个需要反复调试的过程。但是，运行爬虫来检查它是否正常工作意味着每次测试都会多次向服务器发送同样的请求。为了避免这种“不礼貌”的行为，Scrapy 提供了一个名为 `HttpCacheMiddleware` 的内置中间件，可以在 `settings.py` 中用以下方式启用它：

```
HTTPCACHE_ENABLED = True
```

一旦 HTTP 缓存被启用，它将缓存蜘蛛所做的每个请求及相关的响应。所以下一次运行的蜘蛛将不会真正地将请求发送至已经完成请求的服务器。这是一种双赢策略：测试将运行得更快，而对方网站也能节省资源。

#### ➤ 如果可能请使用 HTTP API

许多网站都会提供 HTTP API，以便第三方可以使用它们的数据，而不必抓取它们的网页。在构建网络抓取工具之前，应检查目标网站是否已经提供可以使用的 HTTP API。如果是这样，则使用 API。同样，这也是双赢的：避免挖掘页面的 HTML，并且抓取工具变得更加强大，因为它不需要间接地对内容进行分析。

## 5.2.1 封禁浅析

在很多情况下即使爬虫系统已经做得很有“礼貌”了，但为何仍然会遭遇到被封禁的状况呢？

### 构造合理的 HTTP 请求头

除了处理网站表单，`requests` 模块还是一个设置请求头的利器。HTTP 的请求头是在每次向网络服务器发送请求时传递的一组属性和配置信息。HTTP 定义了十几种“古怪”的请求头类型，不过大多数都不常用。只有下面的七个字段被大多数浏览器用来初始化所有网络请求。





属 性	内 容
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding	gzip, deflate, br
Accept-Language	en-US,en;q=0.9,it;q=0.8,ja;q=0.7,zh-CN;q=0.6,zh;q=0.5,zh-TW;q=0.4,fr;q=0.3
Cache-Control	no-cache
Connection	keep-alive
Host	www.taobao.com
User-Agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36

经典的 Scrapy 爬虫会发送如下请求头:

属 性	内 容
Accept-Encoding	identity
User-Agent	Scrapy/1.1 (+http://scrapy.org)

如果你是一个反爬虫的开发者,你会让哪个请求头访问网站呢?

设置 Scrapy 的默认请求头的最简单办法就是修改 settings.py 文件中的 DEFAULT\_REQUEST\_HEADERS 配置项,具体如下所示。

```
DEFAULT_REQUEST_HEADERS = {
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,image/apng,*/*;q=0.8',
    'Accept-Encoding': 'gzip, deflate, br',
    'Accept-Language': 'en-US,en;q=0.9,it;q=0.8,ja;q=0.7,zh-CN;q=0.6,zh;
q=0.5,zh-TW;q=0.4,fr;q=0.3',
    'Cache-Control': 'no-cache',
    'Connection': 'keep-alive'
}
```

## 超越人类的速度

有一些防护措施完备的网站可能会阻止你快速地提交表单,或者快速地与网站进行交互。即使没有这些安全措施,用一个比普通人快很多的速度从一个网站下载大量信息也可能让自己被网站封杀。



因此，虽然多线程程序可能是一个快速加载页面的好办法——在一个线程中处理数据，另一个线程中加载页面——但是这对编写好的爬虫来说是“恐怖”的策略。还是应该尽量保证一次加载页面且数据请求最小化。

合理控制速度是不应该被破坏的规则。过度消耗别人的服务器资源会让你置身于非法境地，更严重的是这么做可能会把一个小型网站拖垮甚至下线。拖垮网站是不道德的，是彻头彻尾的错误。所以请控制采集速度！

### 设置Cookie的学问

虽然 Cookie 是一把双刃剑，但正确地处理 Cookie 可以避免许多采集问题。网站会用 Cookie 跟踪你的访问过程，如果发现了爬虫异常行为就会中断访问。比如特别快速地填写表单，或者浏览大量页面。虽然这些行为可以通过关闭并重新连接或者改变 IP 地址来进行伪装，但如果 Cookie 暴露了你的身份，再多努力也是白花力气。

在采集一些网站时，Cookie 是不可或缺的。要在一个网站上持续保持登录状态，需要在多个页面中保存一个 Cookie。有些网站不要求在每次登录时都获得一个新 Cookie，只要保存一个旧的“已登录”的 Cookie 就可以访问。

一般在用户登录或者某些操作后，服务端会在返回包中包含 Cookie 信息要求浏览器设置 Cookie，没有 Cookie 会很容易被辨别出来是伪造请求。也有本地通过 JS，根据服务端返回的某个信息处理生成的加密信息设置在 Cookie 中。

### “蜜罐”（honey pot）

如果表单里包含一个具有普通名称的隐含字段（设置蜜罐圈套），比如“用户名”（username）或“邮箱地址”（E-mail address），设计不太好的蜘蛛往往不管这个字段是不是对用户可见，直接填写这个字段并向服务器提交，这样就会中服务器的蜜罐圈套。服务器会把所有隐含字段的真实值（或者与表单提交页面的默认值不同的值）都忽略，而且填写隐含字段的访问用户也可能被网站封杀。

总之，有时检查表单所在的真实页面十分必要，看看有没有遗漏或弄错一些服务器预先设定好的隐含字段（蜜罐圈套）。如果看到一些隐含字段（通常带有较大的随机字符串变量），那么很可能网络服务器会在表单提交时检查它们。另外，还有其他一些检查，用来保证这些当前生成的表单变量只被使用一次还是最近生成的（这样可以避免变量被简单地存储到一个程序中反复使用）。

### 其他因素

- Basic Auth——一般会有用户授权的限制，会在 headers 的 Authentication 字段中要求加入 Basic Auth。





- **Referer**——通常是在访问链接时,必须要带上 **Referer** 字段,服务器会进行验证。例如,抓取京东的评论。
- **User-Agent**——会要求真实的设备,如果不加,则会用编程语言包里自有的 **User-Agent**,可以被辨别出来。
- **Gzip**——请求 **headers** 中带了 **gzip**,返回有时会是 **gzip** 压缩包,需要解压。
- **JavaScript 加密操作**——一般都是在请求的数据包内容中包含一些被 **JavaScript** 加密限制的信息。例如,新浪微博会进行 **SHA1** 和 **RSA** 加密,之前是两次 **SHA1** 加密,发送的密码和用户名都会被加密。

其他字段:因为 **HTTP** 的 **headers** 可以自定义字段,所以第三方可能会加入一些自定义的字段名称或者字段值,这也是需要注意的。

真实的请求过程中,其实不止上面某一种限制,可能是几种限制组合在一起。比如类似 **RSA** 加密,可能先请求服务器得到 **Cookie**,再带着 **Cookie** 去请求服务器拿到公钥,然后用 **JS** 进行加密,最后发送数据到服务器。所以弄清楚其中的原理,并且耐心分析很重要。

## 5.2.2 客户端仿真

一般来说,网站要得知其访客采用的客户端浏览器是什么,最直接的手段就是从请求头中的 **User-Agent** 中读取信息,**User-Agent** 是指包含浏览器信息、操作系统信息等的的一个字符串,也称为一种特殊的网络协议。服务器通过它判断当前访问对象是浏览器、邮件客户端还是网络爬虫。在 **request.headers** 里可以查看 **User-Agent**,关于怎么分析数据包、查看其 **User-Agent** 等信息,这个在前面的章节里提到过。

不要轻易地忽略这个不起眼的 **User-Agent**,因为一旦你忽视它的存在,任意地交由你所使用的工具包或者编程框架来生成,就会轻易地将你的真实身份暴露出来。如果你使用的是 **urllib**,那么默认的 **UA** 就会被设置为 **Python-urllib/2.7**,如果采用 **Scrapy**,则默认的 **UA** 就是 **Scrapy/1.1 (+http://scrapy.org)**,还有些工具包更加“老实”,直接在 **UA** 中带有 **xxx Cralwer** 的字样,这不是明摆着告诉对方:“我是蜘蛛我怕谁?!”

### ➤ 伪装浏览器

可以把 **User-Agent** 的值改为浏览器的方式,甚至可以设置一个 **User-Agent** 池(**list**、**数组**、**字典**都可以),存放多个“浏览器”,每次爬取的时候随机取一个来设置 **request** 的 **User-Agent**,这样 **User-Agent** 会一直变化。

首先打开浏览器,按 **F12** 进入控制台(**Console**),然后输入 **navigator.userAgent**,即可看到 **UA**。例如:



```
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0
```

UA 通常的格式如下:

```
Mozilla/5.0 (平台) 引擎版本 浏览器版本号
```

由于历史上的“浏览器大战”，当时想获得图文并茂的网页，就必须宣称自己是 Mozilla 浏览器。此事导致如今 UA 里通常都带有 Mozilla 字样，最早包含该字样的是 Mozilla/1.0(Win3.1)。现代服务器也不强烈依赖该字符串响应，换言之，现在已经可以不必带上该字样，但几乎每个浏览器依然带有该字样，算是尊重历史吧。

然后是平台部分，这部分可由多个字符串组成，用英文半角分号分开。这部分通常包含操作系统，如果是 Windows 系统，可以参考百度百科 Windows NT 词条。

```
Windows NT 5.0 // 如 Windows 2000
Windows NT 5.1 // 如 Windows XP
Windows NT 6.0 // 如 Windows Vista
Windows NT 6.1 // 如 Windows 7
Windows NT 6.2 // 如 Windows 8
Windows NT 6.3 // 如 Windows 8.1
Windows NT 10.0 // 如 Windows 10
Win64; x64 // Win64 on x64
WOW64 // Win32 on x64
```

其中 WOW64 (Windows-on-Windows 64-bit) 是 Windows 的子系统，让大多数 32 位的程序不用修改也能运行在 64 位系统上。

Linux 系统如下:

```
X11; Linux i686; // Linux 桌面, i686 版本
X11; Linux x86_64; // Linux 桌面, x86_64 版本
X11; Linux i686 on x86_64 // Linux 桌面, 运行在 x86_64 的 i686 版本
```

此外还可以加发行版名: X11、Ubuntu、Linux x86\_64。

macOS (OS X、Mac OS X) 如下:

```
Macintosh; Intel Mac OS X 10_9_0 // Intel x86 或者 x86_64
Macintosh; PPC Mac OS X 10_9_0 // PowerPC
Macintosh; Intel Mac OS X 10.12; // 不用下画线, 用点
```





最后的部分就是系统版本。由于 Mac 的系统多次易名，这里只写出 OS X 和 macOS 的版本号（10.8 版本之后系统名称均为加州景点），分别是：

```
Mountain Lion 10.8.0~10.8.3
Mavericks 10.9.0~10.9.4
Yosemite 10.10.0~10.10.5
EI Capitan 10.11.0~10.11.6
Sierra 10.12.0~10.12.4（至 2017.02，更多的内容可参考维基百科）
```

可指明是 Android、iPod、iPhone、iPad 等：

```
Android; Mobile // Firefox40 及以下
Android; Tablet // Firefox40 及以下
Android 4.4; Mobile // Firefox41 及以上
Android 4.4; Tablet // Firefox41 及以上
iPod touch; CPU iPhone OS 8_3 like Mac OS X
iPhone; CPU iPhone OS 8_3 like Mac OS X
iPad; CPU iPhone OS 8_3 like Mac OS X
```

有时还可能看见加密等级的字符：

```
N; 表示无安全
I; 表示弱安全
U; 表示强安全
```

据 StatCounter 统计，截至 2017 年 1 月，各桌面浏览器的使用分布情况大致如下。

### ➤ Chrome 的 UA

首先是 Google Chrome。以我的浏览器为例：

```
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/56.0.2924.76 Safari/537.36
```

Mozilla/5.0 (Windows NT 10.0; WOW64)，这部分不赘述了。AppleWebKit/537.36 (KHTML, like Gecko) …Safari/537.36，历史上，苹果依靠了 WebKit 内核开发出 Safari 浏览器，WebKit 包含了 WebCore 引擎，而 WebCore 又从 KHTML 衍生而来。由于历史原因，KHTML 引擎需要声明自己是“类似 Gecko”的，因此引擎部分这么写。后来，Google 开发 Chrome 也用了 WebKit



内核，于是也跟着这么写。借用 Littern 的一句话：“Chrome 希望能得到为 Safari 编写的网页，于是决定装成 Safari，Safari 使用了 WebKit 渲染引擎，而 WebKit 又伪装自己是 KHTML，KHTML 又是伪装成 Gecko 的。同时所有的浏览器又都宣称自己是 Mozilla。”。不过，后来 Chrome 28 某个版本改用了 blink 内核，但还是保留了这些字符串。而且，最近的几十个版本中，这部分已经固定，没再变过。

Chrome/56.0.2924.76，这部分才是 Chrome 的版本。56.0 是大版本，2924 是持续增大的一个数字，而 76 则是修补漏洞的小版本。由于没找到版本号的规律，只能寄希望于别人记录了，查找得到如下网站：

- (1) 谷歌 Chrome 旧版本（3~目前最新）。
- (2) Google Chrome（比较新的五六个版本）。
- (3) 下载旧版本 Google Chrome（0.x~46）。

根据上述网站筛选出的数十个版本号，把版本号看作 xx.0.yyyy.zz，通常一个 xx 只对应一个 yyyy，但可能有多个 zz。在不强求正确的情况下，可以随意指定 zz（zz 通常在 0~200 之间），或者都指定为 0，下面为约 20 个大版本。

```
58.0.2995.zz
57.0.2986.zz
56.0.2924.zz
55.0.2883.zz
54.0.2840.zz
53.0.2785.zz
52.0.2743.zz
51.0.2704.zz
50.0.2661.zz
49.0.2623.zz
48.0.2564.zz
47.0.2526.zz
46.0.2490.zz
45.0.2454.zz
44.0.2403.zz
43.0.2357.zz
42.0.2311.zz
41.0.2272.zz
40.0.2214.zz
39.0.2171.zz
```





```
38.0.2125.zz
```

```
37.0.2062.zz
```

### ➤ Firefox的UA

第二部分便是 Firefox。Firefox 的 UA 很容易伪造, 根据 MDN 一篇文章的内容, 格式如下:

```
Mozilla/5.0 (platform; rv:geckoversion) Gecko/geckotrail Firefox/firefoxversion
```

- rv: GeckoVersion 为 Gecko 内核版本号, rv 是 release version 的缩写。最近的几十个版本中, GeckoVersion 和 FirefoxVersion 一致。
- Gecko/GeckoTrail, 桌面端固定不变为 “Gecko/20100101”。
- Firefox/firefoxversion, Firefox 的版本, 形如 xx.0。

不过, 随着 Firefox 更换 Servo 内核的步伐推进, 上述内容可能很快就要发生改变。

### ➤ IE/Edge的UA

第三部分是 IE:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
```

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SV1)
```

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)
```

以上三个都含有 MSIE (Microsoft Internet Explorer), 其中 IE 8 开始加入 Trident 字符串。当使用兼容模式时, UA 如下, 细看可知仅仅只是 MSIE 部分变了:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0)
```

从 IE9 开始, 终于也改为了 “Mozilla/5.0”, 前面这部分没变, 后面可能包含 NET CLR 等内容。

```
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)
```

```
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0)
```

```
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.0; WOW64; Trident/5.0;  
SLCC1; .NET CLR 2.0.50727; Media Center PC 5.0; .NET CLR 3.0.30729)
```

IE10 和 IE9 差不多, 可能包含 NET CLR 等内容:

```
Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0)
```

```
Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0; SLCC2; .NET
```



```
CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0)
```

IE11 看着像是 Gecko 内核 (rv: 11.0)，但显然又不是，同时声明自己是 Trident/7.0 内核。移除了之前版本的“compatible”（兼容）和“MSIE” Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv: 11.0)。IE 继任者 Microsoft Edge 的 UA 格式：

```
Mozilla/5.0 (Windows NT 10.0; &lt;64-bit tags&gt;) AppleWebKit/&lt;WebKit
Rev&gt; (KHTML, like Gecko) Chrome/&lt;Chrome Rev&gt; Safari/&lt;WebKit Rev&gt;
Edge/&lt;EdgeHTML Rev&gt;;&lt;Windows Build&gt;
```

Edge 移除了以下内容：

```
.NET CLR &lt;version&gt;
.NET &lt;version&gt;
TabletPC &lt;version&gt;
Touch
Infopath &lt;version&gt;
Trident &lt;version&gt;
```

以下为按照上述规律伪造的 Win 7 和 Win 10 上 Firefox 和 Chrome 的 UA，共计 66 个。

```
Mozilla/5.0 (Windows NT 6.1; rv:41.0) Gecko/20100101 Firefox/41.0
Mozilla/5.0 (Windows NT 6.1; rv:42.0) Gecko/20100101 Firefox/42.0
Mozilla/5.0 (Windows NT 6.1; rv:43.0) Gecko/20100101 Firefox/43.0
Mozilla/5.0 (Windows NT 6.1; rv:44.0) Gecko/20100101 Firefox/44.0
Mozilla/5.0 (Windows NT 6.1; rv:45.0) Gecko/20100101 Firefox/45.0
Mozilla/5.0 (Windows NT 6.1; rv:46.0) Gecko/20100101 Firefox/46.0
Mozilla/5.0 (Windows NT 6.1; rv:47.0) Gecko/20100101 Firefox/47.0
Mozilla/5.0 (Windows NT 6.1; rv:48.0) Gecko/20100101 Firefox/48.0
Mozilla/5.0 (Windows NT 6.1; rv:49.0) Gecko/20100101 Firefox/49.0
Mozilla/5.0 (Windows NT 6.1; rv:50.0) Gecko/20100101 Firefox/50.0
Mozilla/5.0 (Windows NT 6.1; rv:51.0) Gecko/20100101 Firefox/51.0
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/58.0.2995.0 Safari/537.36
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/57.0.2986.0 Safari/537.36
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/56.0.2924.0 Safari/537.36
```



Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/55.0.2883.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/54.0.2840.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/53.0.2785.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/52.0.2743.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/51.0.2704.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/50.0.2661.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/49.0.2623.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/48.0.2564.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/58.0.2995.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/57.0.2986.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/56.0.2924.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/55.0.2883.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/54.0.2840.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/53.0.2785.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/52.0.2743.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/51.0.2704.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/50.0.2661.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/49.0.2623.0 Safari/537.36

Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like  
Gecko) Chrome/48.0.2564.0 Safari/537.36

Mozilla/5.0 (Windows NT 10.0; WOW64; rv:41.0) Gecko/20100101 Firefox/41.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:42.0) Gecko/20100101 Firefox/42.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:43.0) Gecko/20100101 Firefox/43.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:44.0) Gecko/20100101 Firefox/44.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:45.0) Gecko/20100101 Firefox/45.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:46.0) Gecko/20100101 Firefox/46.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:47.0) Gecko/20100101 Firefox/47.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:48.0) Gecko/20100101 Firefox/48.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:49.0) Gecko/20100101 Firefox/49.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:50.0) Gecko/20100101 Firefox/50.0  
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/58.0.2995.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/57.0.2986.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/56.0.2924.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/55.0.2883.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/54.0.2840.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/53.0.2785.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/52.0.2743.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/51.0.2704.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/50.0.2661.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/49.0.2623.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/48.0.2564.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.2995.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2986.0 Safari/537.36  
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like



```

Gecko) Chrome/56.0.2924.0 Safari/537.36
    Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.0 Safari/537.36
    Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/54.0.2840.0 Safari/537.36
    Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/53.0.2785.0 Safari/537.36
    Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/52.0.2743.0 Safari/537.36
    Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/51.0.2704.0 Safari/537.36
    Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.0 Safari/537.36
    Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/49.0.2623.0 Safari/537.36
    Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/48.0.2564.0

```

## 伪装搜索引擎

告诉你一个非常好的信息——所有的搜索引擎是通过 User-Agent 来告诉网站“我是谁”的。也就是说，我们可以通过设置 User-Agent 来伪装成为网站欢迎的搜索引擎的蜘蛛。

➤ Google的爬虫如下表所示

爬 虫 名	User-agent
Googlebot News	Googlebot-News
Googlebot Images	Googlebot-Image/1.0
Googlebot Video	Googlebot-Video/1.0
Google Mobile (featured phone)	SAMSUNG-SGH-E250/1.0 Profile/MIDP-2.0 Configuration/CLDC-1.1
UP.Browser/6.2.3.3.c.1.101 (GUI)	MMP/2.0 (compatible; Googlebot-Mobile/2.1; +http://www.google.com/bot.html)
Google Smartphone	Mozilla/5.0 (Linux; Android 6.0.1; Nexus 5X Build/MMB29P) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/41.0.2272.96	Mobile Safari/537.36 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)
Google Mobile Adsense	compatible; Mediapartners-Google/2.1; +http://www.google.com/ bot.html
Google Adsense	Mediapartners-Google

续表

爬 虫 名	User-agent
Google AdsBot (PPC landing page quality)	AdsBot-Google (+http://www.google.com/adsbot.html)
Google app crawler (fetch resources for mobile)	AdsBot-Google-Mobile-Apps

#### ➤ 搜狗UA

- Sogou Pic Spider/3.0 (<http://www.sogou.com/docs/help/webmasters.htm#07>)
- Sogou head spider/3.0 (<http://www.sogou.com/docs/help/webmasters.htm#07>)
- Sogou web spider/4.0 (+<http://www.sogou.com/docs/help/webmasters.htm#07>)
- Sogou Orion spider/3.0 (<http://www.sogou.com/docs/help/webmasters.htm#07>)
- Sogou-Test-Spider/4.0 (compatible; MSIE 5.5; Windows 98)

#### ➤ 其他搜索引擎的UA如下表所示

引 擎	爬 虫 名	User-agent
必应	Bingbot	Mozilla/5.0 (compatible; Bingbot/2.0; + <a href="http://www.bing.com/bingbot.htm">http://www.bing.com/bingbot.htm</a> )
雅虎	Slurp	Mozilla/5.0 (compatible; Yahoo! Slurp; <a href="http://help.yahoo.com/help/us/ysearch/slurp">http://help.yahoo.com/help/us/ysearch/slurp</a> )
DuckDuckBot	DuckDuckBot	DuckDuckBot/1.0; (+ <a href="http://duckduckgo.com/duckduckbot.html">http://duckduckgo.com/duckduckbot.html</a> )
百度	Baiduspider	Mozilla/5.0 (compatible; Baiduspider/2.0; + <a href="http://www.baidu.com/search/spider.html">http://www.baidu.com/search/spider.html</a> )
Yandex	YandexBot	Mozilla/5.0 (compatible; Baiduspider/2.0; + <a href="http://www.baidu.com/search/spider.html">http://www.baidu.com/search/spider.html</a> )
脸谱	facebot	facebookexternalhit/1.1 (+ <a href="http://www.facebook.com/externalhit_uatext.php">http://www.facebook.com/externalhit_uatext.php</a> )
Alexa	ia_archiver	ia_archiver (+ <a href="http://www.alexa.com/site/help/webmasters;crawler@alexa.com">http://www.alexa.com/site/help/webmasters;crawler@alexa.com</a> )

当伪装成搜索引擎的爬虫时需要仔细阅读搜索引擎的爬虫说明，因为有些爬虫为了增强自身的辨识度，蜘蛛采用了固定 IP+User-agent 来共同标识。

### Scrapy中的客户端仿真

客户端仿真说起来复杂但实现起来并不难，其原理不过是从一个装有一堆 User-Agent 的数组中随机选出其一，然后添加到请求头中。Scrapy 也提供了一个 UserAgentMiddleware 中间



件用于设置 User-Agent, 但它却没有多大用处, 因为它只能设置一个 User-Agent, 一旦设置, 所有的蜘蛛都只会采用这个 User-Agent, 完全不能达到随机伪装的效果。

首先新建一个 ua.py 文件, 只定义一个 user\_agents 的数组变量, 将需要模仿的 User-Agent 列表存入其中, 具体代码如下:

```
user_agents = [  
    'Mozilla/5.0 (Windows NT 6.1; rv:41.0) Gecko/20100101 Firefox/41.0',  
    'Mozilla/5.0 (Windows NT 6.1; rv:42.0) Gecko/20100101 Firefox/42.0',  
    'Mozilla/5.0 (Windows NT 6.1; rv:43.0) Gecko/20100101 Firefox/43.0',  
    ...  
    ## 省略  
]
```

然后创建一个 RandomUserAgentMiddleware 中间件:

```
from scrapy import signals  
from ua import user_agents  
import random  
  
class RandomUserAgentMiddleware(object):  
  
    def process_request(self, request, spider):  
        request.headers.setdefault(b'User-Agent', random.choice(user_agents))
```

最后在 settings.py 中启用这个蜘蛛中间件:

```
SPIDER_MIDDLEWARES={  
    'myproject.RandomUserAgentMiddleware':800  
}
```

这样就能支持随机的 User-Agent 了, 当需要时就可以修改 ua.py 中的数组内容以增减各种 User-Agent 的字符串。

## 5.2.3 化身万千——蜘蛛世界的易容术

建立网络爬虫的第一原则是: 所有信息都可以伪造。可以用非本人的邮箱发送邮件, 通过命令行自动化鼠标的行为, 或者通过浏览器耗费网站流量来吓唬网管。

但是有一件事情是不能作假的，那就是你的 IP 地址。从技术的角度说，IP 地址是可以通过发送数据包进行伪装的，这就是分布式拒绝服务攻击技术(Distributed Denial of Service, DDoS)，攻击者不需要关心接收的数据包（这样发送请求时就可以使用假 IP 地址）。但是网络数据采集是一种需要关心服务器响应的行为，所以我们认为 IP 地址是不能造假的。

阻止网站被采集的焦点主要集中在识别人类与机器人的行为差异上。封杀 IP 地址这种矫枉过正的行为，就好像是农民不靠喷农药给庄稼杀虫，而是直接用火烧来彻底解决问题。它是最后一步棋，不过是一种非常有效的方法，只要忽略危险 IP 地址发来的数据包就可以了。但是，使用这种方法会遇到以下几个问题。

### IP地址访问列表很难维护

虽然大多数大型网站都会用自己的程序自动管理 IP 地址访问列表（机器人封杀机器人），但是至少需要人偶尔检查一下列表，或者至少要监控问题的增长。因为服务器需要根据 IP 地址访问列表去检查每个准备接收的数据包，所以检查接收数据包时会额外增加一些处理时间。多个 IP 地址乘以海量的数据包会使检查时间呈指数级增长。为了降低处理时间和处理复杂度，管理员通常会对 IP 地址进行分组管理并制定相应的规则。如果这组 IP 中有一些危险分子，就“把这个区间的所有 256 个地址全部封杀”。于是产生了下一个问题。

### 封杀IP地址可能会导致意外后果

这种一刀切的行为可能直接导致与网站本身链接的某些资源被屏蔽而变得不可访问，对于网站本身来说是一种重大的资源损失。

### IP可以更换

封杀 IP 对于具有固定 IP 的服务器或者云来说是毁灭性的，但对于个人来说却毫无作用。在用 28.8KB “猫”拨号上网的年代，只要断开网络重新连接一次 IP 地址就会自动更换，直到今天用的“光猫”上 ADSL 还是一样的。即使 IP 被封杀了，换个 IP 就没事了。

虽然有这些缺点，但封杀 IP 地址依然是一种常用的手段，服务器管理员用它来阻止可疑的网络爬虫入侵服务器。

## 5.2.3.1 使用代理

使用代理进行爬网是隐蔽自身真正 IP 地址，使蜘蛛的行踪变得神出鬼没的最佳办法。接下来详细地讲述之前提到的几个库（如 urllib、requests、selenium 及主打的 Scrapy 框架）如何使用代理发送 Web 请求的方法和具体代码的写法。

### urllib

首先以基础的 urllib 为例，看一下代理的设置方法，代码如下：



```
from urllib.error import URLError
from urllib.request import ProxyHandler, build_opener

proxy = '127.0.0.1:9743'
proxy_handler = ProxyHandler({
    'http': 'http://' + proxy,
    'https': 'https://' + proxy
})
opener = build_opener(proxy_handler)
try:
    response = opener.open('http://httpbin.org/get')
    print(response.read().decode('utf-8'))
except URLError as e:
    print(e.reason)
```

运行结果如下:

```
{
  "args": {},
  "headers": {
    "Accept-Encoding": "identity",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "Python-urllib/3.6"
  },
  "origin": "106.185.45.153",
  "url": "http://httpbin.org/get"
}
```

在这里我们需要借助于 `ProxyHandler` 设置代理, 参数是字典类型, 键名为协议类型, 键值是代理。注意此处代理前面需要加上协议, 即 HTTP 或者 HTTPS。此处设置了 HTTP 和 HTTPS 两种代理, 当我们请求的链接是 HTTP 协议时, 它会调用 HTTP 代理; 当请求的链接是 HTTPS 协议时, 它会调用 HTTPS 代理, 所以此处生效的代理是 `http://127.0.0.1:9743`。

创建完 `ProxyHandler` 对象之后, 我们需要利用 `build_opener()` 方法传入该对象来创建一个 `Opener`, 这样就相当于此 `Opener` 已经设置好代理了。接下来直接调用它的 `open()` 方法即可使用此代理访问我们想要的链接。

运行输出结果是一个 JSON，它有一个 `origin` 字段，标明了客户端的 IP。验证一下此处的 IP，确实为代理的 IP，而并不是真实的 IP。这样就成功设置好代理，并可以隐藏真实 IP 了。

如果遇到需要认证的代理，则可以用如下方法进行设置：

```
from urllib.error import URLError
from urllib.request import ProxyHandler, build_opener

proxy = 'username:password@127.0.0.1:9743'
proxy_handler = ProxyHandler({
    'http': 'http://' + proxy,
    'https': 'https://' + proxy
})
opener = build_opener(proxy_handler)
try:
    response = opener.open('http://httpbin.org/get')
    print(response.read().decode('utf-8'))
except URLError as e:
    print(e.reason)
```

这里改变的只是 `proxy` 变量，只需要在代理前面加入代理认证的用户名密码即可。其中 `username` 就是用户名，`password` 为密码。例如，`username` 为 `foo`，密码为 `bar`，那么代理就是 `foo:bar@127.0.0.1:9743`。

如果代理是 Socks5 类型，则可以用如下方式来设置代理：

```
import socks
import socket
from urllib import request
from urllib.error import URLError

socks.set_default_proxy(socks.SOCKS5, '127.0.0.1', 9742)
socket.socket = socks.socksocket
try:
    response = request.urlopen('http://httpbin.org/get')
    print(response.read().decode('utf-8'))
except URLError as e:
    print(e.reason)
```





此处需要一个 Socks 模块，可以通过如下命令进行安装：

```
$ pip install PySocks
```

本地有一个 Socks5 代理，运行在 9742 端口，运行成功之后和上文 HTTP 代理的输出结果是一样的：

```
{
  "args": {},
  "headers": {
    "Accept-Encoding": "identity",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "Python-urllib/3.6"
  },
  "origin": "106.185.45.153",
  "url": "http://httpbin.org/get"
}
```

结果的 origin 字段同样为代理的 IP，设置代理成功。

## requests

对于 requests 来说，代理设置更加简单，我们只需要传入 proxies 参数即可。

还是以上例中的代理为例，我们来看一下 requests 的代理的设置：

```
import requests

proxy = '127.0.0.1:9743'
proxies = {
    'http': 'http://' + proxy,
    'https': 'https://' + proxy,
}

try:
    response = requests.get('http://httpbin.org/get', proxies=proxies)
    print(response.text)
except requests.exceptions.ConnectionError as e:
    print('Error', e.args)
```



运行结果:

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"
  },
  "origin": "106.185.45.153",
  "url": "http://httpbin.org/get"
}
```

可以发现 requests 的代理设置比 urllib 简单很多,只需要构造代理字典即可,然后通过 proxies 参数即可设置代理,不需要重新构建 Opener。

其运行结果的 origin 也是代理的 IP,证明代理已经设置成功。

如果代理需要认证,则同样在代理的前面加上用户名和密码,代理的写法就变成:

```
proxy = 'username:password@127.0.0.1:9743'
```

和 urllib 一样,只需要将 username 和 password 替换即可。

如果需要使用 Socks5 代理,则可以使用如下方式:

```
import requests

proxy = '127.0.0.1:9742'
proxies = {
    'http': 'socks5://' + proxy,
    'https': 'socks5://' + proxy
}

try:
    response = requests.get('http://httpbin.org/get', proxies=proxies)
    print(response.text)
except requests.exceptions.ConnectionError as e:
    print('Error', e.args)
```





在这里需要额外安装一个 Socks 模块, 命令如下:

```
$ pip install "requests[socks]"
```

运行结果是完全相同的:

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"
  },
  "origin": "106.185.45.153",
  "url": "http://httpbin.org/get"
}
```

另外还有一种设置方式, 和 urllib 中的方法相同, 使用 Socks 模块, 也需要像上面一样安装该库, 设置方法如下:

```
import requests
import socks
import socket

socks.set_default_proxy(socks.SOCKS5, '127.0.0.1', 9742)
socket.socket = socks.socksocket

try:
    response = requests.get('http://httpbin.org/get')
    print(response.text)
except requests.exceptions.ConnectionError as e:
    print('Error', e.args)
```

这样也可以设置 Socks5 代理, 运行结果完全相同。相比第一种方法, 此方法是全局设置, 不同情况下可以选用不同的方法。



## PhantomJS

对于 PhantomJS，代理设置方法可以借助于 `service_args` 参数，也就是命令行参数，代理设置方法如下：

```
from selenium import webdriver

service_args = [
    '--proxy=127.0.0.1:9743',
    '--proxy-type=http'
]

browser = webdriver.PhantomJS(service_args=service_args)
browser.get('http://httpbin.org/get')
print(browser.page_source)
```

在这里我们只需要使用 `service_args` 参数，将命令行的一些参数定义为列表，在初始化时传递即可。

如果需要认证，那么只需要再加入 `--proxy-auth` 选项即可，这样参数就改为：

```
service_args = [
    '--proxy=127.0.0.1:9743',
    '--proxy-type=http',
    '--proxy-auth=username:password'
]
```

将 `username` 和 `password` 替换为认证所需的用户名和密码即可。

## Scrapy

Scrapy 提供了一个下载器中间件 `scrapy.downloadermiddlewares.HttpProxyMiddleware`，用于设置爬网的代理，只要在 `settings.py` 文件中设置以下几个配置项就可以启用该代理中间件：

```
DOWNLOADER_MIDDLEWARES={
    'scrapy.downloadermiddlewares.HttpProxyMiddleware':900
}

HTTPPROXY_ENABLED = True
```

`HttpProxyMiddleware` 中间件并不可以在设置文件中直接指定代理地址，它设置代理地址





的方法有两种, 一种是从系统代理中读取, 另一种就是在生成请求实例 `request` 后在 `request.meta['proxy']` 中设置代理地址。也就是说, 要么在蜘蛛的 `start_requests` 方法中进行设置, 要么编写一个下载器中间件并将其实现优先级设置得比 `HttpProxyMiddleware` 更高, 在自定义中间件中先设定好 `request.meta['proxy']` 的值。以下代码是采用自定义中间件以设置 HTTP 代理的具体做法:

```
class ProxyMiddleware(object):

    def process_request(self, request, spider):
        request.meta['proxy'] = "http://61.129.70.131:8080"
```

为了让代码变得更为灵活, 可以将代理的设置放到配置文件中。假定将代理地址的配置项命名为 `PROXY_ADDRESS`, 配置文件将修改成以下的样子:

```
DOWNLOADER_MIDDLEWARS={
    'my_crawler.middlewares.ProxyMiddleware':800,
    'scrapy.downloadermiddlewares.HttpProxyMiddleware':888
}

HTTPPROXY_ENABLED = True
PROXY_ADDRESS = 'http://61.129.70.131:8080'
```

然后将中间件代码调整一下, 向构造函数传入初始化代理地址:

```
class ProxyMiddleware(object):

    def __init__(self, proxy_address):
        self.proxy_address = proxy_address

    @classmethod
    def from_crawler(cls, crawler):
        if not crawler.settings.get('PROXY_ADDRESS'):
            raise NotConfigured
        return cls(PROXY_ADDRESS)

    def process_request(self, request, spider):
        request.meta['proxy'] = self.proxy_address
```



同时采用两个中间件的做法可以用更少的代码来“改造”HttpProxyMiddleware，增加ProxyMiddleware就能直接在外部配置中更改代理地址。

### 5.2.3.2 代理池

Scrapy 的爬网性能是极高的，前文中已提及如果能让爬虫的行为变得更像正常人，就要将 Scrapy 的并发能力与爬网速度降低。事实上，如果需要每天爬取巨量的信息，则这样低效的爬网能力是无法让人忍受的。既要保持性能，又要保持隐蔽性，最佳办法就是采用随机 UA 配合随机代理，这样就可以让爬虫拥有不同的 IP，那么反爬系统就无法察觉了。

随机代理的做法非常简单，我们只将代理的地址保存到一个 Python 数组中，然后在中间件中随机选取一个即可。具体代码如下：

```
# proxies.py

proxies = [
    '223.150.219.96:8888',
    '219.138.58.26',
    '122.114.31.177:808',
    '219.138.58.143',
    # ... 省略
]
```

在中间件中导入上述数组：

```
import random
from proxies import proxies

class RandomProxyMiddleware(object):

    def process_request(self, request, spider):
        request.meta['proxy'] = random.choice(proxies)
```

这种做法有一个缺点，就是很多代理地址都会经常失效，而且如果都放在一个文件中，靠人工更新，则基本上是没有实际使用意义的。在“中级虫术”中曾举了一个某代理网站爬虫的示例，现在我们可以将它改造一下，把爬取的 IP 地址与有效时间写入 Redis 中。因为 Redis 可以设置每条记录的有效时间，时间一到记录就会被自动删除。这样我们就不需要人工来更新这个 IP 池了，而是由爬虫在某一时间到某代理网站爬取一次来自动更新 IP 池即可。





首先为某代理网站爬虫编写一个管道，将数据保存到 Redis 中。由于该网站的代理数据非常多，我们取其中可连接时间最长、速度最快的即可。所以先编写一个用于过滤数据的管道，代码如下所示。

```
class ProxyFilter(object):
    def process_item(self, item, spider):
        if item['speed'] > 2000 or item['ttl'] < 1800000 or
item['connection_time'] > 2000:
            raise DropItem(u'筛除低质量代理%s' % item)
        else:
            return item
```

接下来编写 Redis 的存储管道，这里采用 Redis 的集合类型，以下是 Redis 的存储管道的代码：

```
from scrapy.exceptions import DropItem
from redis import StrictRedis
import datetime

class RedisWriter(object):
    def __init__(self):
        self.redis = StrictRedis(port=6379)

    def process_item(self, item, spider):
        item_data = dict(item)
        ip = item_data.pop('ip')

        ttl = item['ttl'] / 1000

        item_val = '%s:%s:%s' % (item['protocol'], ip, item['port'])

        # 将数据写入集合中
        self.redis.sadd('proxies', item_data, ex=ttl)
```

将以上两个管道加入爬虫项目，然后运行一次。这样我们就建立了一个常用的高速代理 IP 池。

接下来编写一个随机读取 Redis 代理池中 IP 的中间件，只要将上面的随机代理的中间部分



改造一下即可，具体代码如下：

```
from redis import StrictRedis

class RandomProxyMiddleware(object):

    def __init__(self):
        self.redis = StrictRedis(port=6379)

    def process_request(self, request, spider):
        # 从 Redis 集合中随机选出一个代理
        request.meta['proxy'] = self.redis.srandmember()
```

### 5.2.3.3 Tor网络

洋葱路由（The Onion Router）网络是一种 IP 地址匿名手段，常用缩写为 Tor。由网络志愿者服务器构建的洋葱路由器网络，通过不同服务器构成多个层（就像洋葱一样），把客户端包在最里面。数据进入网络之前会被加密，因此任何服务器都不能偷取通信数据。另外，虽然每一个服务器的入站和出站通信都可以被查到，但是要想查出通信的真正起点和终点，必须要知道整个通信链路上所有服务器的入站和出站通信细节，而这基本是不可能实现的。

Tor 不仅可以提供客户端的匿名访问，还可以提供服务器的匿名服务。通过使用 Tor 网络，用户可以维护位置不可知的服务器。这些服务器所构成的网络被称为“Tor Hidden Services”，一般的互联网则相应地被称为明网。因为在明网中，客户端和服务端彼此知道对方的真实 IP 地址，而在 Tor Hidden Services 中，双方不知道对方的 IP 地址。若服务端能做到不记录用户使用信息，以及客户端能做到任何时刻都不输入真实个人数据，则通过 Tor 隐藏服务可以达成上网的完全匿名性。

如果要访问 Tor 隐藏服务，则客户端必须安装 Tor 浏览器，在搭载 Android 操作系统的手机或平板电脑上，必须安装 Orfox。

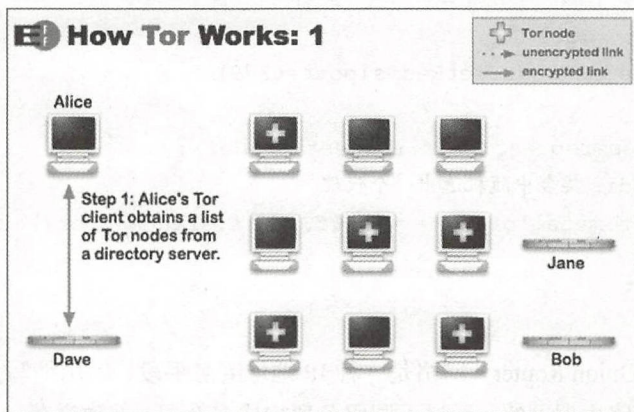
在 Tor 浏览器中，在地址栏输入 Tor 隐藏网络特有的顶级域名.onion，可以访问 Tor 隐藏服务。Tor 浏览器可以识别.onion 域名，并自动路由到隐藏的服务。然后，隐藏的服务将请求交由标准的服务器软件进行处理，这个服务器软件应该预先进行配置，从而只侦听非公开的接口。

Tor 隐藏服务有个好处，由于不需要公开的 IP 地址，服务可以躲在防火墙和 NAT 背后。但如果这个服务还可以通过一般的互联网（明网）来访问，则会受到相关联的攻击，这样就没有真正隐藏起来。

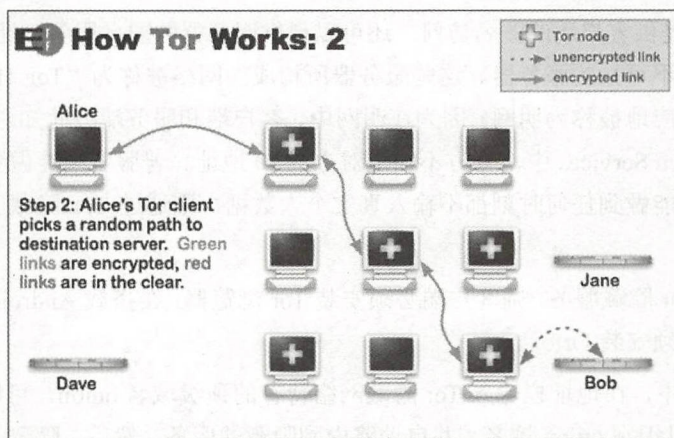


## Tor的工作原理

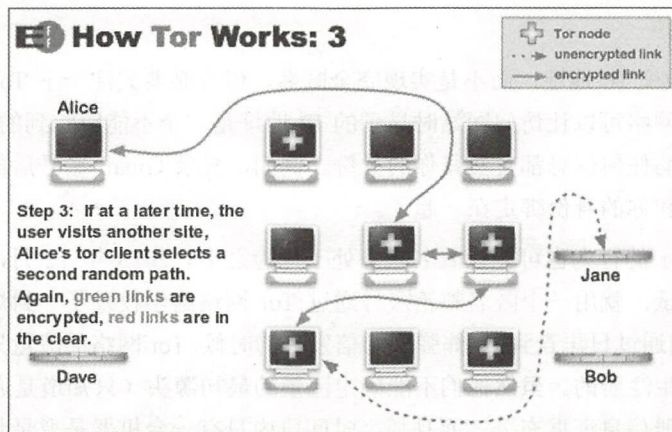
为了能充分地了解 Tor 网络, 接下来以 Tor 官网上一个例子来说明 Tor 的工作原理。首先假定 Alice 要通过 Tor 网络访问 Bob 的服务器, 那么 Alice 会先从 Dave 的服务器(目录服务)上获取一份 Tor 节点的清单, 如下图所示。



然后, Tor 网络会随机为 Alice 在网络中选择一条访问路径, 最后到达 Bob 的机器上, 如下图所示。



而当其他客户反过来要访问 Alice 的机器时, Tor 客户端同样会随机地选择另一条网络路径达到 Alice, 如下图所示。



如此一来，数据包在 Tor 网络中的传输几乎是随机行为，非常难以跟踪。

Tor隐藏服务的例子（在非Tor的浏览器输入.onion网址是无意义的）

- Tor 隐藏维基 (<http://zqkltwi4fecvo6ri.onion/>)：列出常用的隐藏网站，以及使用隐藏服务的技巧和注意事项。也有明网的地址 (<https://thehiddenwiki.org/>)，可使用一般的浏览器访问。
- SIGAINT：标榜隐私与安全，广受资安人士信赖的电子邮件服务。
- DuckDuckGo：主要运作在明网的搜索引擎，标榜隐私、不记录用户信息和搜索历史（相反，Google、Yahoo、Bing、蕃薯藤等都会记录），也提供其他服务 (<http://3g2upl4pq6kufc4m.onion/>)，让用户更彻底地匿名化。
- Facebook 虽然禁止 Tor 用户注册账号，却提供了.onion 的隐藏服务，供已注册的用户连接 (<https://www.facebookcorewwi.onion/>)。

### 基于Tor匿名网络的多IP爬虫

更换 IP 的方式有多种，其中 Tor 类型适合 IP 更换次数不大、网页数据量也不大，但是又厌恶代理天天失效的场景——使用 Tor 在本机搭建一个出口端口，让需要更换 IP 的爬虫程序制定 proxies 指向的端口。可使用的 IP 池子总数为 1000 左右，实际有 500 左右可以使用，匿名性当然不用质疑了。

Tor 的部署成本非常小，只要本机能够访问谷歌即可拥有 500 个 IP 供使用，并且能够保证相当高的匿名性。但是问题也是存在的，如果目标网站网页内容多，或者在抓取时使用 PhantomJS 等方式，则一样会遇到网速的限制，此时就需要另外的方式了。



## 匿名的局限性

Tor 的目的是改变 IP 地址,而不是实现完全匿名,但有必要关注一下 Tor 匿名方法的能力和不足。虽然 Tor 网络可以让访问网站时显示的 IP 地址是一个不能跟踪到的 IP 地址,但是在网站上留给服务器的任何信息都会暴露你的身份。例如,登录 Gmail 账号后再用 Google 搜索,那些搜索历史就会和你的身份绑定在一起了。

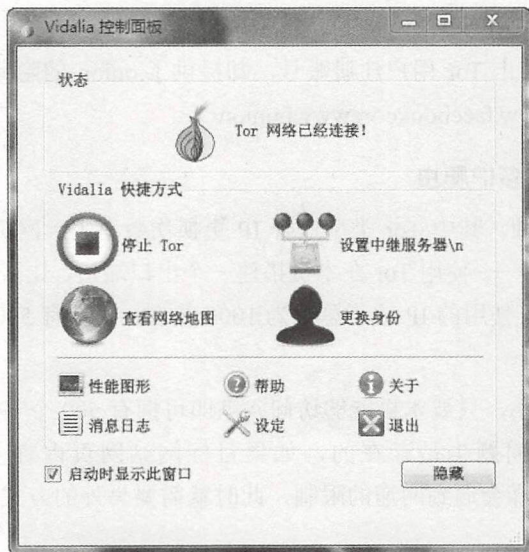
另外,登录 Tor 的行为也可能让匿名状态处于危险之中。2013 年 12 月,一个哈佛大学本科生想逃避期末考试,就用一个匿名邮箱账号通过 Tor 网络给学校发了一封炸弹威胁信。结果哈佛大学的 IT 部门通过日志查到,在炸弹威胁信发来的时候,Tor 网络的流量只来自一台机器,而且是一个在校学生注册的。虽然他们不能确定流量的最初源头(只知道是通过 Tor 发送的),但是作案时间和注册信息证据充分,而且那个时间段内只有一台机器是登录状态,这就有充分理由起诉那个学生了。

登录 Tor 网络不是一个自动的匿名措施,也不能让你进入互联网上的任何区域。虽然它是一个实用的工具,但是用它的时候一定要谨慎、清醒,并且遵守道德规范。

## 安装Tor网络

如果要在 Windows 上使用 Tor,则直接去 Tor 的官网下载安装洋葱浏览器就可以了。当然,进行爬虫开发只需要安装 Tor 核心就可以了。这里有两个版本 Tor 控制器,Windows 的 Vidalia 和 OS X 版本的 Arm (Anonymizing Relay Monitor)。

Windows 上的 Vidalia 如下图所示。



### ➤ 配置Tor

接下来重点讲述如何在 macOS 上配置 Tor 网络，在 macOS 上需要先使用 HomeBrew 安装 Tor，具体如下所示。

```
$ brew install tor
```

安装完成后要启用 Tor 服务：

```
$ brew services restart tor
```

然后配置 Tor 网络的控制密码，这个密码用作连接到该服务端口的其他客户端程序（比如 Python 程序）验证之用。生成密码的指令如下：

```
$ tor -hash-password mypassword
```

当生成该密码后，Tor 指令实质上是更改了其配置文件，可以在以下位置找到该文件：

```
$ cat /usr/local/etc/tor/torrc
```

该文件只有三个配置项：

```
ControlPort 9051
```

```
HashedControlPassword
```

```
16:3DEC46BBE972692D60750B9C5230A1068C71210CA5F643DA518E67D554
```

由于我们需要通过前置代理，所以增加一些其他配置项，上述文件内加入以下配置信息：

```
Socks5Proxy 127.0.0.1:1086
```

```
# 连接国外 VPN 代理的地址与端口
```

```
CookieAuthentication 1
```

```
# 启用 Cookie 验证
```

重启 Tor 使配置生效：

```
$ brew services restart tor
```

Tor 的启动是比较慢的，使用 HomeBrew 命令启动只是将服务加载到内存中运行，而 Tor 是否正常运行尚未可知。最佳的检验办法是先不要启动 Tor，直接在命令行运行 Tor 命令，观察配置是否能真正生效，如下图所示。



```

RayOSX:tor Ray$ tor
Jan 04 00:05:14.215 [notice] Tor 0.3.0.10 (git-c33db290a9d8d0f9) running on Darwin with Libevent 2.1.8-stable, Open
SSL 1.0.2l and Zlib 1.2.11.
Jan 04 00:05:14.216 [notice] Tor can't help you if you use it wrong! Learn how to be safe at https://www.torproject
.org/download/download#warning
Jan 04 00:05:14.216 [notice] Read configuration file "/usr/local/etc/tor/torrc".
Jan 04 00:05:14.219 [warn] ControlPort is open, but no authentication method has been configured. This means that
any program on your computer can reconfigure your Tor. That's bad! You should upgrade your Tor controller as soon
as possible.
Jan 04 00:05:14.219 [notice] Opening Socks listener on 127.0.0.1:9000
Jan 04 00:05:14.219 [notice] Opening Control listener on 127.0.0.1:9051
Jan 04 00:05:14.000 [notice] Parsing GEOIP IPv4 file /usr/local/Cellar/tor/0.3.0.10/share/tor/geoip.
Jan 04 00:05:14.000 [notice] Parsing GEOIP IPv6 file /usr/local/Cellar/tor/0.3.0.10/share/tor/geoip6.
Jan 04 00:05:14.000 [notice] Bootstrapped 0%: Starting
Jan 04 00:05:14.000 [notice] Starting with guard context "default"
Jan 04 00:05:14.000 [notice] Bootstrapped 45%: Asking for relay descriptors
Jan 04 00:05:17.000 [notice] Bootstrapped 50%: Loading relay descriptors
Jan 04 00:06:21.000 [notice] New control connection opened from 127.0.0.1.
Jan 04 00:06:29.000 [notice] Bootstrapped 57%: Loading relay descriptors
Jan 04 00:06:34.000 [notice] Bootstrapped 62%: Loading relay descriptors
Jan 04 00:07:07.000 [notice] Bootstrapped 68%: Loading relay descriptors
Jan 04 00:07:34.000 [notice] Bootstrapped 75%: Loading relay descriptors
Jan 04 00:07:56.000 [notice] Bootstrapped 80%: Connecting to the Tor network
Jan 04 00:07:56.000 [notice] Bootstrapped 85%: Finishing handshake with first hop
Jan 04 00:07:57.000 [notice] Bootstrapped 90%: Establishing a Tor circuit
Jan 04 00:07:59.000 [notice] Tor has successfully opened a circuit. Looks like client functionality is working.
Jan 04 00:07:59.000 [notice] Bootstrapped 100%: Done

```

如果出现 Tor 的引导启动能运行到 100%，则表明已经成功接入 Tor 网络。

### ➤ 安装 Privoxy

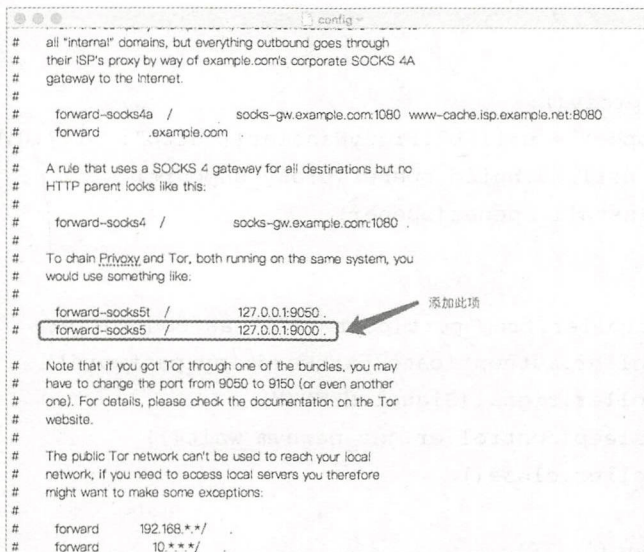
安装控制器的目的是能让 Python 在爬虫内连接到 Tor 网络，但上述配置过程只能通过 Socks5 进行连接，而爬虫是通过 HTTP 连接的，因此我们还需要将 Socks5 转换成为 HTTP 连接。此时就需要安装 Privoxy 来进行转换。输入以下指令进行安装：

```
$ brew install Privoxy
```

Privoxy 是一款不进行网页缓存且自带过滤功能的代理服务器，针对 HTTP、HTTPS 协议。通过其过滤功能，用户可以保护隐私、对网页内容进行过滤、管理 Cookie，以及拦阻各种广告等。Privoxy 可以单机使用，也可以应用到多用户的网络。它也可以与其他代理相连，更可以突破互联网审查。

安装完成后用文本编辑器打开 Privoxy，对其配置进行修改：

```
$ open /usr/local/etc/privoxy/config
```



配置完成后关闭并保存，然后通过 HomeBrew 启动 Privoxy:

```
$ brew services start privoxy
```

此时将连接 Tor 网络的代理换成本地的 8118 端口，也就是 127.0.0.1:8118，这样就完成 Socks5 到 HTTP 的转换了。

### ➤ Scrapy实现Tor网络的多IP爬虫

当使用前面讲述的办法成功配置 Tor 网络和 Privoxy 之后，如果直接用浏览器访问 <http://ifconfig.me/ip>，则会发现无论怎么刷新，IP 都不会变。你可能会有这样的疑问：“不是说好有 500 个 IP 可以更换的吗？”确实，Tor 的 IP 是会变的，但并不是自动的，而是要通过指令，这才是为何要用 Python 进行控制的原因。

首先安装 stem 工具包，在 Python 中连接 Tor 控制器，安装方法如下：

```
$ pip install stem
```

接下来写一段代码来测试用 stem 是否能控制 Tor:

```

from stem import Signal
from stem.control import Controller
import urllib2
import time

```



```

def _set_urlproxy():
    proxy_support = urllib2.ProxyHandler({"http": "127.0.0.1:8118"})
    opener = urllib2.build_opener(proxy_support)
    urllib2.install_opener(opener)

def new_ip():
    with Controller.from_port(port=9051) as controller:
        controller.authenticate(password='my_password')
        controller.signal(Signal.NEWNYM)
        time.sleep(controller.get_newnym_wait())
        controller.close()

for i in range(0, 10):
    new_ip()
    _set_urlproxy()
    print urllib2.urlopen("http://icanhazip.com/").read()

```

上述代码中最关键的部分是调用 `controller.signal(Signal.NEWNYM)` 向 Tor 发出更换 IP 的请求, 这样每次发出的请求都会获得一个随机的新 IP。

### ➤ 基于Tor网络的多IP爬虫

将爬虫系统接入 Tor 网络的办法其实通过上文中对 Tor 的配置应该有一定的了解, 在讲述这个示例之前先来总结一下:

- (1) 配置 Tor 网络。
- (2) 配置 Privoxy。
- (3) 通过 Python 设置上网的代理端口, 并且每个请求都会更换一次 IP。

我们只需要编写一个下载器中间件, 在 `process_request` 处理之前设置好代理并更换新的 IP 就能实现目标了, 具体实现代码如下:

```

class TorProxyMiddleware(object):

    def __init__(self, http_proxy=None, tor_control_port=None, tor_password=
None):

```

```

    if not http_proxy:
        raise Exception('http proxy setting should not be empty')

    if not tor_control_port:
        raise Exception('tor control port setting should not be empty')

    if not tor_password:
        raise Exception('tor password setting should not be empty')

    self.http_proxy = http_proxy
    self.tor_control_port = tor_control_port
    self.tor_password = tor_password
    self.count = 1
    self.times = 50

    @classmethod
    def from_crawler(cls, crawler):
        http_proxy = crawler.settings.get('HTTP_PROXY')
        tor_control_port = crawler.settings.get('TOR_CONTROL_PORT')
        tor_password = crawler.settings.get('TOR_PASSWORD')

        return cls(http_proxy, tor_control_port, tor_password)

    def process_request(self, request, spider):
        self.count = (self.count + 1) % self.times
        if not self.count:
            # access tor ControlPort to signal tor get a new IP
            with Controller.from_port(port=self.tor_control_port) as controller:
                controller.authenticate(password=self.tor_password)
                controller.signal(Signal.NEWNYM)

        # scrapy support http proxy
        request.meta['proxy'] = self.http_proxy

```

如果单纯地更换 IP 而继续使用默认的 UA 那就真的是前功尽弃了，所以在配置中加入了客户端仿真中的 RandomUserAgentMiddleware 产生随机 UA，让随机 IP 与随机 UA 相结合才能实现高度的仿真效果，具体配置如下：



```

DOWNLOADER_MIDDLEWARES = {
    'example.middlewares.RandomUserAgentMiddleware': 543,
    'example.middlewares.TorProxyMiddleware': 544,
}

# Pirvoxy listening on 8118
HTTP_PROXY = 'http://127.0.0.1:8118'
# Tor ControlPort
TOR_CONTROL_PORT = 9052
# Tor ControlPort authorization password
TOR_PASSWORD = '123456'

```

在实际使用中, Tor 的最大优点就是部署成本非常低, 关键在于对 Tor 客户端的配置, 还有就是对 VPN 的要求会高一些。如果将爬虫直接部署在国外的服务器上, 则可以完全不依赖于 VPN 来做前置代理, 性能还可以相对地提高一些。总的来说, 基于 Tor 网络的多 IP 爬虫在多种高隐匿爬虫方案中最具有实效性。

#### 5.2.3.4 ADSL服务器

Tor 网络爬虫非常适应于 Linux 和 macOS 环境, 如果将其应用到 Windows 中几乎就是一条艰难的入坑之路。在 Windows 上更为常见的多 IP 爬虫方案是购买一台 ADSL 服务器, 原理很简单, 在家庭网络中宽带上网只要断开再拨号一次, 连接成功就会更换一次外网 IP, 而且连接建立后网速比较稳定——这就是动态 IP 了, 这个 IP 池很大, 一个城市一般会有 5 万~30 万的 IP, 基本上是用不完的。所以只要有一台接入了宽带的计算机, 都可以叫作 ADSL 动态 IP 服务器。但是, 有部分时间会消耗在网络建立上(大约十秒)。

通过指令不停地断开 ADSL 连接后重新连接, 以此方式来更换新的 IP, 几乎没有任何技术含量可言。

了解了原理, 实现 ADSLMiddleware 中间件就非常容易了, 以下是全部代码:

```

import os

class AdslMiddleware(object):

    def __init__(self):
        self.name = g_adsl_account["ADSL_NAME"]
        self.username = g_adsl_account["ADSL_USER"]
        self.password = g_adsl_account["ADSL_PWD"]

```

```
def connect(self):
    cmd_str = "rasdial %s %s %s" % (self.name, self.username, self.password)
    os.system(cmd_str)
    time.sleep(5)

def disconnect(self):
    cmd_str = "rasdial %s /disconnect" % self.name
    os.system(cmd_str)
    time.sleep(5)

def reconnect(self):
    self.disconnect()
    self.connect()

def process_request(self, request):
    reconnect()
    return request;
```

这种方案只适用于 Windows 平台，因为只有在 Windows 平台上有 rasdial 命令。此方案很“暴力”也很有效，与 Tor 相比根本没有什么部署成本，唯一的成本就是需要购置 ADSL 的云服务器。

## 5.2.4 反跟踪

要从服务器端主动跟踪和分析客户端的异常行为，其实手段相当有限。只要深入地反思一下这个问题就能找到答案：“你可以用何种手段在网站上跟踪访客？”首先我们知道 Web 服务器是无状态的，要跟踪访客的行为就需要知道客户端的某一种状态。服务端只能从请求头上得知客户端的一些基本信息。例如，通过 UserAgent 获取浏览器的名称和版本，通过 IP 地址计算归属地，通过 Referrer 了解这个请求是从哪个页面引入的，通过查询字符串（Query String）显式地传入参数进行某些判断。而这些都是一次性信息，并不带有任何状态。

### Cookie跟踪

要模拟出状态化的效果只能通过 Cookie 或者服务端会话（session）实现，如果开启真正的服务端会话，则会使服务器的容量到达低谷。因为一旦开启会话，服务器会为每个访问的客户开启一个专用的空间来保存会话。会话一般以 20 分钟为可用时长，而一旦访问量过大，会话空



间就有可能拖垮服务器。禁用会话是开发高性能网站的基本常识,因此很多开发语言中的会话功能都是一种“伪会话”,大多都是通过向 Cookie 写入一个会话 ID 来进行识别的。这样说来, Cookie 也就成为了唯一能跟踪客户行为的手段。

比起会话 ID,用户的登录信息才是 Cookie 的常客,几乎所有的用户登录功能都要用 Cookie 来存储。如果网站开发者重视安全性,则往往会对 Cookie 中存储的值进行非对称加密,这样就会让客户端无法伪造 Cookie 中的值。

那么在爬虫中应该如何应对呢?

### ➤ 禁用Cookie

通过禁止 Cookie,这是客户端主动阻止服务器写入。禁止 Cookie 可以避免那些采用 Cookies 识别爬虫的网站的封杀。在 Scrapy 爬虫中可以设置 `COOKIES_ENABLED=False`,即不启用 Cookies middleware,不向 Web Server 发送 Cookies。

这种方法直接、粗暴、简单。但也遇到过有些网站会发送随机 Cookie,如果检测到客户端没有回发随机生成的 Cookie,则将客户端认定为爬虫而直接封掉。另外,这种做法一旦遇到需要登录的场景就完全失效,所以说它仅仅适用于那些不需要验证登录身份的开放式网站或者页面。

### ➤ 合并Cookie

也就是将每次响应收到的 Cookie 先存起来,在下次发出请求时使用。这种做法在 Scrapy 中是不用设置的,正如前面所述 Cookies 中间件是默认启用的,它会自动进行处理。

### referrer

referrer 用于告诉服务器当前这个请求是由哪个页面转入的,这是一个 URL 值,经常用于反盗链检测(是反盗链而不是反爬)。这种手段常见于一些大型网站或者老牌网站(如百度),一旦触发了反盗链机制,所获得的网页内容就与原来的内容完全不同了,但爬虫却可能完全不知。它不会使请求失败,而是会写入其他版权保护信息。例如,爬取的是一张图片,但由于 referrer 的设置违反了对方的引用策略,那么得到的可能就是一个“该图片来自 XXX 网站”的占位图。

referrer 策略包含以下值:

- no-referrer——最简单的策略是“no-referrer”,表示所有的请求都不带 referrer。
- no-referrer-when-downgrade——主要针对于受 TLS 保护的 URL(如 HTTPS),简单地说就是在 HTTPS 的页面中,如果连接的资源也是 HTTPS 的,则发送完整的 referrer,如果连接的资源是 HTTP 的,则不发送 referrer。这是在没有特别指定 referrer 策略时浏览器的默认行为。

- same-origin——对于同源的链接，会发送 `referrer`，其他的不会。同源意味着域名需要相同，`example.com` 和 `not.example.com` 是非同源的。
- origin——这个策略对于任何资源来说只发送源的信息，不发送完整的 URL。
- strict-origin——这个策略类似于 `origin` 和 `no-referrer-when-downgrade` 的合体，如果一个 HTTPS 页面中链接到 HTTP 的页面或资源，则不会发送 `referrer`。HTTP 页面链接和 HTTPS 链接到 HTTPS 都只发送来源页面的源信息。
- origin-when-cross-origin——该策略在同源的链接中发送完整的 URL，其他情况仅发送源信息。相同的域名，HTTP 和 HTTPS 协议被认为是非同源的。
- strict-origin-when-cross-origin——对于同源请求，发送完整的 URL；对于同为 HTTPS 的，只发送源信息；对于 HTTPS 页面，只发送源信息；HTTPS 页面中的 HTTP 请求不发送 `referrer`。
- unsafe-url——主要解决 HTTPS 页面中的 HTTP 资源不发送 `referrer` 的问题，它会使在 HTTPS 页面中的 HTTP 资源发送完整的 `referrer`。
- 空字符串——空字符串表示没有 `referrer` 策略，默认为 `no-referrer-when-downgrade`。

那么怎么才知道目标网站采用了哪个 `referrer` 策略呢？

`referrer` 策略会通过以下方法声明：

- (1) 通过 HTTP 请求头中的 `Referrer-Policy` 字段。
- (2) 通过 `meta` 标签，`name` 为 `referrer`，如 `<meta name="referrer" content="same-origin" />`。
- (3) 通过 `<a>`、`<area>`、`<img>`、`<iframe>`、`<link>` 元素的 `referrerpolicy` 属性。
- (4) 通过 `<a>`、`<area>`、`<link>` 元素的 `rel=noreferrer` 属性。
- (5) 通过隐式继承。

因此，一旦遇到具有反盗链策略的网站，就要先对上述地方进行检测，得出对方的引用策略后才能正确应对。Scrapy 提供了一个 `RefererMiddleware` 中间件用于处理 `referrer`，但它只能用于统一地将所有请求设置为配置中指定的 `referrer` 策略，默认情况下它是被启用的。如果要关闭它，则只需要将配置文件中的 `REFERER_ENABLED` 设置为 `False` 即可。

```
REFERER_ENABLED=False
```

通过设置 `REFERRER_POLICY` 可以默认启用 `referrer` 策略：

```
REFERRER_POLICY='origin'
```



另外,还可以在代码中动态设置 `referrer` 策略:

```
request.meta['referrer_policy'] = 'origin-when-cross-origin'
```

## 5.2.5 绕开蜜罐

蜜罐(Honeypots)本质上是一种对攻击方进行欺骗的技术,通过布置一些作为诱饵的主机、网络服务或者信息,诱使攻击方对它们实施攻击。从而可以对攻击行为进行捕获和分析,了解攻击方使用的工具与方法,推测攻击意图和动机,就能够让防御方清晰地了解他们所面对的安全威胁,并通过技术和管理手段来增强实际系统的安全防护能力。

蜜罐好比是情报收集系统。蜜罐是引诱别人实施攻击的目标,引诱黑客前来攻击。攻击者入侵后,就可以知道攻击者是如何得逞的,随时了解针对服务器发动的最新的攻击和漏洞。还可以通过窃听黑客之间的联系,收集黑客所用的种种工具,并且掌握他们的社交网络。

### 蜜罐是一种思想

在军事领域,以响尾蛇导弹为代表的红外制导武器,可以利用红外探测器捕获和跟踪目标自身辐射的能量来追击目标。对于飞机来说,没有任何方法能在空中关停产生热辐射的发动机。所以,对飞机来说,响尾蛇导弹是一种非常可怕的敌人。

为了避免被响尾蛇导弹击落,很多飞机都会配备一种防御武器:红外诱饵弹。在响尾蛇导弹接近自己时,放出红外诱饵弹,吸引对方去攻击诱饵弹,从而达到让自己金蝉脱壳的作用。

类似的事情也发生在互联网安全领域。一个接入互联网的网站,只要能和外部产生通信,就有被黑客攻击的可能——就像飞机在空中无法消除发动机产生的热辐射一样。但是网站能像飞机发射红外诱饵一样,用某种陷阱来引诱攻击者,实现自身不被攻击的目的。这种引诱黑客攻击的“陷阱”就是“蜜罐”。从广义上看,“蜜罐”并不具体指某种技术,而是一种思想。

### 蜜罐的成本

既然蜜罐是一种思想指导,也就是说,它是没有具体实现标准的。但蜜罐的实现与设计可以反过来给予爬虫系统以启发。在我过去参与过的诸多 Web 项目中,应用蜜罐技术的地方并不多,而且对于一个实际 Web 项目来说,实现用户需求和提高使用体验比耗费人力与时间在网页上增加几个不知何时会触发的陷阱要重要得多。如果客户没有明确地提出这方面的要求,则项目组根本不会“多此一举”。

蜜罐也是一项需要开发、需要设计与调试、需要人工与时间成本的功能。对于一些中小企业而言,蜜罐只不过是“纸上谈兵”的东西而已。真正用到它的更多会是那些以数据为生的大

型或者超大型的企业或网站，它们为了捍卫自己的数据安全，会用尽一切办法与爬虫或者各种与之不利的网络机器人做斗争。

网络服务提供商经常会使用到蜜罐或者密网技术，他们用蜜罐来防范那些传统的攻击。对于 Web 而言，只要页面可以访问就是不设防的，或者说是防不胜防，要将蜜罐部署到所有的页面相当于希特勒要防御盟军从法国海岸线登陆一样困难。

## 反爬虫蜜罐的实现

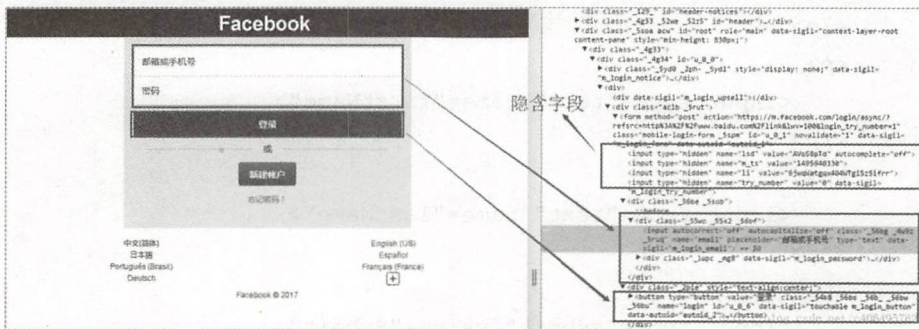
### ➤ 触发性蜜罐

顾名思义，触发性蜜罐就是指爬虫达到了蜜罐的触发条件，蜜罐就会打开。最典型的用法就是在表单中加入隐藏的输入字段，如果是人访问网页，则这些网页元素是不会对浏览器造成任何影响的，甚至可以说客户根本不知道当前网页上存在这些东西。但是，如果爬虫没有识别出这些隐藏的输入字段的作用，为了得到数据，任意地向服务器提交输入字段，那披在爬虫身上的伪装可能马上被服务器识别出来，对方服务器可能会做出以下反应：

- 拒绝处理请求，阻止提交表单。这种方式比较常见，在 Web 端是最容易实现的一种。
- 保持静默，返回一个重定向地址将爬虫引导至新的蜜罐之中进行二重身份确认。
- 直接封杀 IP——这种方式比较极端，因为很容易导致“错杀”。
- 如果爬虫带有实名信息，比如登录后的 Cookies，那可以处理的空间就更大了。

在 HTML 表单中，“隐含”字段可以让字段的值对浏览器可见，但是对用户不可见（除非查看网页源代码）。随着越来越多的网站开始用 Cookie 存储状态变量来管理用户状态，在找到另一个最佳用途之前，隐含字段主要用于阻止爬虫自动提交表单。

下图显示的例子就是 Facebook 登录页面上的隐含字段。虽然表单里只有三个可见字段（username、password 和一个确认按钮），但是在源代码中表单会向服务器传送大量的信息。



如果提交时这个值不在表单处理页面上，服务器就有理由认为这个提交不是从原始表单页面上提交的，而是由一个网络机器人直接提交到表单处理页面的。绕开这个问题的最佳方法就



是, 首先采集表单所在页面上生成的随机变量, 然后提交到表单处理页面。

接下来举一个简单的例子, 这个例子中的网页只有一个表单, 其他内容先忽略:

```
<html>
<head>
  <title>蜜罐网页</title>
<style>
body {
  overflow-x:hidden;
}
.hidden {
  position:absolute;
  right:50000px;
}
</style><style type="text/css"></style></head>

<body>
  <h2>蜜罐</h2>
  <a href="http://example.com/details/4" style="display:none;">点击此处
</a>
  <a href="http://example.com/more">了解更多</a>
  <form>
    <input type="hidden" name="phone" value="valueShouldNotBeModified">
    <p>
      <input type="text" name="email" class="hidden" value=
"intentionallyBlank">
    </p>
    <p>
      <input type="text" name="firstName">
    </p>
    <p>
      <input type="text" name="lastName">
    </p>
    <p>
      <input type="submit" value="Submit">
    </p>
  </form>
```

```
</body>
</html>
```

这三个元素通过三种不同的方式对用户隐藏：

- 第一个链接通过简单的 CSS 属性设置 `display:none` 进行隐藏。
- 电话号码字段 `name="phone"` 是一个隐含的输入字段。
- 邮箱地址字段 `name="email"` 是将元素向右移动 50000 像素（应该会超出显示器的边界）并隐藏滚动条。

由于 `CrawlSpider` 这类爬虫并不清楚目标网页存放在网站的哪一层深度，因此会从入口页一次性获取所有的链接，然后一个页面一个页面地大规模寻找。对于网站本身而言，这种爬虫是最不受欢迎的，它们就像是冲进文明世界的野蛮人和劫匪，在城市中强占道路、横冲直撞，明目张胆地抢劫或者偷窃。对于什么地方都敢闯入的野蛮人，最简单的办法就是在他们的前进方向中设置陷阱，用简单的方法解决简单的问题。

`Selenium` 可以获取访问页面的内容，所以它可以区分页面上的可见元素与隐含元素。通过 `is_displayed()` 可以判断元素在页面上是否可见。以下是一个检测网页上元素是否可见，并以此作为对蜜罐进行简单判断的程序，具体代码如下所示。

```
# -*- coding:UTF-8 -*-
# check_honeypots.py

from selenium import webdriver

if __name__ == '__main__':
    url = 'http://www.facebook.com' # 更改此 URL 可以检测不同的网页
    driver = webdriver.PhantomJS()
    driver.get(url)
    links = driver.find_elements_by_tag_name('a')

    for link in links:
        if not link.is_displayed():
            print('链接:' + link.get_attribute('href') + ', 可能是一个圈套。')

    fields = driver.find_elements_by_tag_name('input')

    for field in fields:
        if not field.is_displayed():
            print('输入字段' + field.get_attribute('name') + '是一个隐含的输入
            字段。')
```



### ➤ 缠杀型蜜罐

这种蜜罐有点“杀敌一千自损八百”的意思，但比上一种以触发陷阱的方式更“智能”一些，已经具有互动式蜜罐的形式了。蜜罐一旦判定访客为爬虫，会先为爬虫提供一个新的 URL 作为入口，像上面描述的那样将其引诱进去，然后向爬虫发送特定的响应内容。但这种内容是真还是假就难说了，它的目的只是拖垮爬虫系统！主要有以下两种手段：

让响应进程睡眠：

这种做法非常简单，就是让进程“sleep”一下，这个间隔是长短不一的，取决于开发者的喜好。这样做可以拖慢爬虫的速度，甚至可能让蜘蛛的响应超时。应对这种蜜罐得多注意爬网时蜘蛛的超时数量，当出现大量超时的情况时，蜘蛛很有可能进入蜜罐之中了。

在 Scrapy 中有一个 `scrapy.downloadermiddlewares.DownloadTimeoutMiddleware` 的下载器中间件，用于设置爬虫的超时时间，但不能总认为超时就是进入蜜罐了，某些情况下有可能是 DNS 解释造成的。Scrapy 也考虑到了这个因素，Scrapy 的 DNS 默认超时是 60 秒，可以通过 `DNS_TIMEOUT` 来设置，也可以设置 `DownloadTimeoutMiddleware` 的超时时间。

```
DOWNLOAD_TIMEOUT = 180 # 默认是 180 秒
```

造成爬虫进入死循环：

更难缠的蜜罐就是让爬虫进入一个死循环之中，它们会在响应正文中附带大量可见的循环链接，先让爬虫误判，然后继续深入爬取数据，最后令其进入深入的“跟进”链接的死循环内不能自拔。即使爬虫做了 URL 的去重操作，只要链接的 URL 不重复但内容重复，爬虫还是会陷入内容循环，不停地工作，导致系统在“正常中中断”。

解决这个问题有两个方案。

方案 1：在 `setting.py` 中，设定爬虫的嵌套次数（链接层次的深度）上限（全局设定，实际是通过 `DepthMiddleware` 实现的）。

```
DEPTH_LIMIT = 20
```

方案 2：在 `parse` 方法中，通过读取 `response` 的深度在代码级进行判断。

```
def parse(self, response):  
    if response.meta['depth'] > 100:  
        print 'Loop?'
```

要绕开反爬虫的蜜罐并不是很难的事，更不会是一个复杂的技术问题。只要对爬取的目标不采用盲目的、扫荡式的方式，在爬取目标网页前多做一些分析工作，这些蜜罐是很容易被发

现且轻松避开的。那会不会遇到那些具有“智能”型分析能力且又是组合型的蜜罐呢？这是有可能的，但要看正在爬取的目标网站是否愿意为了保护数据付出如此巨大的成本代价。简言之，就是先从常理进行分析，再考虑技术的复杂性与可能性，这样可以防止让我们的思路进入“思想蜜罐”。

### 小结

本节的真正意图就是说明如何与反爬机制进行斗争，也就是所谓的反反爬网。

其实这一切回归本源就是一种拟人化，所谓拟人化就是让蜘蛛的爬取行为让对方网站看起来更像是一个真实的人在操作。反爬是一个斗智斗力的过程，不可能在一本书中尽述，简单地说是这一系列行为而做出的策略与判断。

而且，要更像人可能还需要付出一些性能代价。例如，一个完整的网页除了网页正文，还有网络资源，比如 JS、CSS、图片、媒体文件等。虽然蜘蛛只需要在网页正文中获取所需要的资源就行了，但是对方网站可能做出这样的判断策略：“如果客户端只对网页正文发出请求而从来不下载该网页的其他资源时则认为其是爬虫”。

## 5.3 虫海

“大巧不攻，以力破法”

制造一个成本极高而无用的天才，不如制造一堆廉价而有用的白痴。

本节的重点：

- (1) 简单的爬虫框架。
- (2) scrapy-redis 的使用。

- 安装；
- 配置；
- 编写基于 scrapy-redis 的爬虫；
- 编写基于 scrapy-redis 的管道。

- (3) Scrapy Celery 的分布式架构。

### 5.3.1 分布式爬虫架构

Scrapy 并没有提供内置的机制来支持分布式（多服务器）爬取。不过还是有办法进行分布



式爬取, 这取决于要怎么“分布”了。如果有很多 Spider, 那分布负载最简单的办法就是启动多个 Scrapy, 并分配到不同机器上。

如果想要在多个机器上运行一个单独的 Spider, 则可以将要爬取的 URL 进行分块, 并发送给 Spider。

首先, 准备要爬取的 URL 列表, 并分配到不同文件的 URL 中:

```
http://somedomain.com/urls-to-crawl/spider1/part1.list
http://somedomain.com/urls-to-crawl/spider1/part2.list
http://somedomain.com/urls-to-crawl/spider1/part3.list
```

接着在 3 个不同的 Scrapy 服务器中启动 Spider。Spider 会接收一个 (Spider) 参数 part, 该参数表示要爬取的分块:

```
$ curl http://scrapy1.mycompany.com:6800/schedule.json -d project=myproject
-d spider=spider1 -d part=1
$ curl http://scrapy2.mycompany.com:6800/schedule.json -d project=myproject
-d spider=spider1 -d part=2
$ curl http://scrapy3.mycompany.com:6800/schedule.json -d project=myproject
-d spider=spider1 -d part=3
```

这种分布式爬虫之间形成的是一个对等网, 每个爬虫节点之间没有从属关系, 其最大的作用是将爬网对等化, 通过控制清单来分配爬网任务。优点是可以将现有的 Scrapy 项目直接部署, 无须多加改动。但缺点也是显而易见的——任务列表需要人工分配与更新, 可以适用于一些非持久性的轻度增量式爬网场合。

## 5.3.2 认识 scrapy-redis

scrapy-redis 是一个基于 Redis 的 Scrapy 分布式组件。它利用 Redis 对用于爬取的请求 (requests) 进行存储和调度 (schedule), 并对爬取产生的项目 (Items) 存储以供后续处理使用。scrapy-redis 重写了 Scrapy 中比较关键的代码, 将 Scrapy 变成了一个可以在多个主机上同时运行的分布式爬虫。

### scrapy-redis 分布式原理

假设有 4 台不同操作系统的计算机: macOS、Ubuntu 14.04、Ubuntu 16.04、CentOS 7.2, 任意一台计算机都可以作为 Master 端或 Slaver 端。比如:

- Master 端（核心服务器）——使用 Ubuntu 16.04，搭建一个 Redis 数据库，不负责爬取，只负责 URL 指纹判重、request 的分配，以及数据的存储。
- Slaver 端（爬虫程序执行端）——使用 macOS、Ubuntu 14.04、CentOS 7.2，负责执行爬虫程序，运行过程中提交新的 request 给 Master。

首先 Slaver 端从 Master 端“拿”任务（request、URL）进行数据抓取，Slaver 抓取数据的同时产生新任务的 request 以便提交给 Master 处理。

Master 端只有一个 Redis 数据库，负责将未处理的 request 进行去重和任务分配，将处理后的 request 加入待爬队列，并且存储爬取的数据。

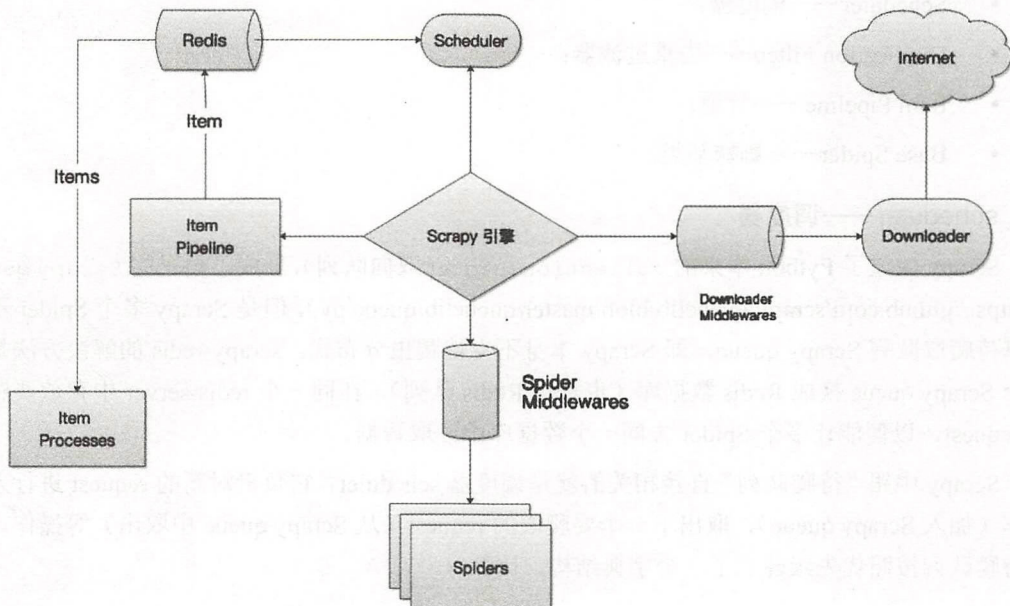
scrapy-redis 默认使用的就是这种策略，实现起来很简单。因为任务调度等工作 scrapy-redis 都已经帮我们做好了，我们只需要继承 RedisSpider、指定 redis\_key 就行了。

缺点是，scrapy-redis 调度的任务是 request 对象，里面的信息量比较大（不仅包含 URL，还有 callback 函数、headers 等信息），可能会降低爬虫速度，而且会占用 Redis 大量的存储空间。所以如果要想保证效率，那么就需要一定硬件水平。

### scrapy-redis的分布式架构

scrapy-redis 是一双飞翔的翅膀，它自身并没有爬虫能力，它仅仅是为 Scrapy 框架披上了一层铠甲。所以阅读本节需要对 Scrapy 的工作原理有一定的了解，如果不感兴趣则可以跳过。

加上 scrapy-redis 之后，架构如下图所示。





如果我们自行编写一个分布式爬虫在多台主机上运行,则需要将爬虫的爬取队列进行共享。也就是说,每台主机都需要访问一个共享的队列,然后爬虫从队列中取一个 request 进行爬取。当然这些 scrapy-redis 都已经帮我们做好了,需要做的是如下操作:

- 初始化爬虫,创建一个 Redis 的客户端,连接 Redis。
- 查看请求队列是否为空,如果是空则等待,当请求的队列不为空时,从请求队列中拿出一个 request。
- 获得 request,经过 scheduler 调度后,Engine 会将 request 取出,送给 downloader 进行请求。
- 经过请求后,返回给 Engine,Engine 将结果返回给用户写的爬虫,对结果进行处理。可能出现下一个 request,也可能是 Item。
- 如果请求后得到的是一个 request,则会通过 scheduler 再次调度,判断 request 是否重复,并将 request 放入请求队列。
- 如果已经得到了 Item,则 Scrapy 会将 Item 交给 pipeline 处理。

可见,scrapy\_redis 就是将 request 调度的队列、请求的队列和获取的 Item 放在了一个多台主机可以同时访问的 Redis 的数据结构中。

scrapy-redis 提供了下面四种组件(components)——四种组件意味着这四个模块都要做相应的修改:

- Scheduler——调度器;
- Duplication Filter——去重过滤器;
- Item Pipeline——管道;
- Base Spider——蜘蛛基类。

### scheduler——调度器

Scrapy 改造了 Python 本来的 `collection.deque`(双向队列),形成了自己的 Scrapy queue (<https://github.com/scrapy/queuelib/blob/master/queuelib/queue.py>),但是 Scrapy 多个 Spider 不能共享待爬取队列 Scrapy queue,即 Scrapy 本身不支持爬虫分布式。scrapy-redis 的解决方法是把这个 Scrapy queue 换成 Redis 数据库(也是指 Redis 队列),在同一个 redis-server 中存放要爬取的 request,以便能让多个 Spider 去同一个数据库中读取数据。

Scrapy 中跟“待爬队列”直接相关的就是调度器 scheduler,它负责对新的 request 进行入列操作(加入 Scrapy queue),取出下一个要爬取的 request(从 Scrapy queue 中取出)等操作。它把待爬队列按照优先级建立了一个字典结构,比如:

```
{
    优先级 0 : 队列 0
    优先级 1 : 队列 1
    优先级 2 : 队列 2
}
```

然后根据 request 中的优先级来决定该入哪个队列，出列时则按优先级较小的优先出列。为了管理这个比较高级的队列字典，scheduler 需要提供一系列的方法。但是原来的 scheduler 已经无法使用，所以使用 scrapy-redis 的 scheduler 组件。

### Duplication Filter——去重过滤器

在 Scrapy 中用集合实现 request 去重功能，Scrapy 中把已经发送的 request 指纹放入一个集合中，把下一个 request 的指纹拿到集合中比对，如果该指纹存在于集合中，说明这个 request 发送过了，如果没有则继续操作。

在 scrapy-redis 中去重是由 Duplication Filter 组件实现的，它通过 Redis 的 set 数据类型的不重复特性，巧妙地实现了 Duplication Filter 去重。scrapy-redis 调度器从引擎接受 request，并将 request 的 URL 指纹存放于 Redis 的 set 中，由其自身的类型检查是否重复，并将不重复的 request push 写入 Redis 的 request queue。

引擎请求 request（Spider 发出的）时，调度器从 Redis 的 request queue 队列里根据优先级“pop”出一个 request 返回给引擎，引擎将此 request 发给 Spider 进行处理。

### Item Pipeline——管道

引擎将（Spider 返回的）爬取到的 Item 发给 Item Pipeline，scrapy-redis 的 Item Pipeline 将爬取的 Item 存入 Redis 的 Items queue。

修改过的 Item Pipeline 可以很方便地根据 key 从 Items queue 中提取 Item，从而实现 Items processes 集群。

### Base Spider——蜘蛛基类

不再使用 Scrapy 原有的 Spider 类，重写的 RedisSpider 继承了 Spider 和 RedisMixin 两个类，RedisMixin 是用来从 Redis 中读取 URL 的类。

当生成一个 Spider 继承 RedisSpider 时，调用 setup\_redis 函数，这个函数会去连接 Redis 数据库，然后设置 signals（信号）：

- 一个是 Spider 空闲时的 signal，会调用 spider\_idle 函数，这个函数调用 schedule\_next\_request 函数，保证 Spider 是一直活跃的状态，并且抛出 DontCloseSpider 异常。



- 一个是抓到 Item 时的 signal, 会调用 item\_scraped 函数, 这个函数会调用 schedule\_next\_request 函数, 获取下一个 request。

### 5.3.3 示例: 分布式电商爬虫

在这个示例中我将会重拾中级虫术中的电商爬虫并将其重构, 使之支持分布式爬取。由于我在“处理 JavaScript”一节中采用了 Selenium 与 Splash 两种实现方式, 如果要改造成为分布式爬虫, 则采用 Selenium 会比使用 Splash 显得更容易。因为 Splash 需要 Web 服务端支持, 所以如果要改造为分布式结构, 则每个爬虫部署节点都需要配置一个 Splash 服务器, 这样无疑会增大节点部署的复杂性。再者, 使用 Selenium+PhantomJS 会有更高的隐秘性, 虽然 Selenium+PhantomJS 架构会比 Splash 架构的情况略低, 但俗话说得好, “好汉架不住人多”, 分布式爬虫拼的就是数量。

#### 安装及环境配置

在安装 scrapy-redis 之前需要先准备 Redis 服务器或者虚拟机, 以提供 Master 任务队列服务。然后输入以下指令来安装 Redis 工具包与 scrapy-redis。

```
$ pip install redis
$ pip install scrapy-redis
```

#### 配置

首先, 要在 settings.py 中添加 Redis 的配置:

使用 REDIS\_URL 声明 Redis 服务的访问信息:

```
# Redis 设置
REDIS_URL = 'redis://username:***@xxx.xxx.xxx.xxx:6379/0'
```

或者, 采用另一种配置方式:

```
REDIS_HOST = "xxx.xxx.xxx.xxx"      # REDIS 主机
REDIS_PORT = 6379                    # REDIS 服务端口
REDIS_PASSWD = "****"               # 访问密码
REDIS_DB = 0                         # 数据库索引号
```

**注:** REDIS\_URL 的优先级最高, 大于 REDIS\_\*, 两种配置方式只要配置一种就好了。

然后，配置 scrapy-redis：

```
#使用 scrapy-redis 中的去重组件
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"
# 使用 scrapy-redis 中的调度器
SCHEDULER = "scrapy_redis.scheduler.Scheduler"
# 允许暂停后能保存进度
SCHEDULER_PERSIST = True

# 指定排序爬取地址时使用的队列
# 默认的，按优先级排序（Scrapy 默认），由 sorted set 实现的一种非 FIFO、LIFO 方式
SCHEDULER_QUEUE_CLASS = 'scrapy_redis.queue.SpiderPriorityQueue'
# 可选的，按先进先出排序（FIFO）
# SCHEDULER_QUEUE_CLASS = 'scrapy_redis.queue.SpiderQueue'
# 可选的，按后进先出排序（LIFO）
# SCHEDULER_QUEUE_CLASS = 'scrapy_redis.queue.SpiderStack'

ITEM_PIPELINES = {
    'scrapy_redis.pipelines.RedisPipeline': 400
}
```

这里最重要的一点是配置 scrapy-redis 的 Scheduler，这可以说是 scrapy-redis 的动作核心。在上述配置中出现了一个去重过滤器，这是 scrapy-redis 原生搭载的基于 Redis 的去重过滤器。既然是使用分布式爬虫，那么目标数据一定极为庞大，所以推荐使用在“高效的 Redis 布隆过滤器”一节中提到的布隆过滤器。最后启用 RedisPipeline，将采集后的数据进行存储等后处理。

## 中间件

在“将爬虫接入 Selenium”一节中采用了下载器中间件 SeleniumMiddler 令爬虫系统能对 JavaScript 进行前置处理，在本示例中仍然会使用此中间件，此处就不表述了。

### ➤ 随机UA中间件

这个中间件在“突破封印”一节中已详细介绍过，此处是将 UA 的列表直接写入 settings.py 配置中，具体代码如下所示。

```
# -*- coding: utf-8 -*-
import random
```



```

from .settings import USER_AGENTS

# 随机的 User-Agent
class RandomUserAgent(object):
    def process_request(self, request, spider):
        userAgent = random.choice(USER_AGENTS)
        request.headers.setdefault("User-Agent", userAgent)

```

### 蜘蛛的改写

scrapy\_redis 的唯一缺陷就是不能通过中间件技术接入爬虫系统，所有基于 scrapy\_redis 架构下的蜘蛛都必须继承自 scrapy\_redis.spiders.RedisSpider 类。

另外，在蜘蛛中我们将初始 URL 存放在 start\_urls 类成员中。而在 RedisSpider 类中，我们需要初始化的成员是 redis\_key，这是 Redis 数据库的一个队列的名。爬虫开始运行时，通过读入 settings.py 中的 Redis 配置来访问远程的 Redis 数据库。如果根据 redis\_key 从数据库中获取初始的 URL 来爬取，则 Redis 数据库所在的主机就是 Master，所有从机（Slaver）都配置好主机的 Redis 信息，然后运行爬虫，就能不断地从 Redis 数据库中获取待爬取的 URL。

以下是改写后的蜘蛛代码：

```

from ..items import ProductItem
from ..product_data import product_sns # 导入货号
import urllib

from scrapy_redis.spiders import RedisSpider
from redis import Redis

class TBItemSpider(RedisSpider):
    name = 'TB'
    allowed_domains = ['电商网址']
    redis_key = 'spider:start_urls'

    def parse(self, response):
        # 原有方法内容不变，此处省略...

```

如果仔细阅读以上代码，你一定会问：“那谁来产生 start\_urls 的爬网地址呢？”。就以上述代码为例，存储在 Redis spider:start\_urls 键内的值是由外部程序产生的，是不是很奇

怪？习惯于单机开发的模式之后一下子很难将思路切换过来，由于蜘蛛代码是被分布在各个节点上的，而且是处于持久运行的状态，只有侦测到 Redis 中的 `spider:start_urls` 有新的 URL 值的时候才会运行起来，如果在蜘蛛中写入产生 URL 的逻辑，那么每个副本节点在运行的时候都会向 Redis 写入相同的 URL，这样就乱套了。所以只能将写入 `start_urls` 的过程分离到爬虫系统之外，或者是某个脚本代码，又或者是某个 Web 进程。

在本例中我们就将 URL 的生成逻辑保存在一个 `gen_urls.py` 的 Python 脚本中，在任何能访问 Redis 的机器上运行该文件就可以启动整个分布式爬虫网络。

```
redis = Redis(host=REDIS_HOST,
              port=REDIS_PORT,
              db=REDIS_DB,
              password=REDIS_PASSWD)

base_url = 'https://s.taobao.com/search?q=%s'

if __name__ == '__main__':
    for sn in product_sns:
        keyword = u'匡威%s' % sn
        url = self.base_url % urllib.quote(keyword.encode('utf-8'))
        redis.lpush('spider:start_urls', url) # 写入 Redis 启动分布式爬虫网络
```

分布式爬虫的开发就这么简单？对！将前文内容与本节示例归纳一下，可以得到以下几点：

- (1) 部署 Redis 服务。
- (2) 定义 Item。
- (3) 继承 `RedisSpider` 编写 `Spider`。  
定义 `redis_key`。
- (4) 在 `settings.py` 中加入 `scrapy_redis` 要求的配置项。
- (5) 将爬虫部署在多台节点上（对于持续迭代的项目建议使用 `Scrapyd` 部署）。
- (6) 编写生成起始爬取 URL 的启动脚本。

## 5.4 可视化爬虫

Portia 是 Scrapinghub 旗下的一个很出色的产品，它是一个基于 Web 的可以快速生成 Scrapy 项目代码的工具。Portia 是一个开源项目，可以到 <https://github.com/scrapinghub/portia> 上获取它





的源代码，或者在 Scrapinghub 的官网上直接使用它提供的在线服务。

Portia 可以方便地在 Web 界面上直接生成项目、蜘蛛、Item，还可以通过选中目标网页元素来生成 xpath 和 parse 方法，大大地简化与加速了 Scrapy 项目的开发。既然有这么好的工具，为什么前面一直不使用它甚至都没有提及，而是在全书的最后一个章节才介绍呢？原因在于如果没有透彻地了解并掌握爬虫技术的方方面面，而急于使用 Portia 这类旨在简化编程的工具，则会让我们对细节一无所知。相反，如果已经对 Scrapy 了如指掌，那么 Portia 会成为你的一大臂助。

Portia 提供的界面功能都是基于 Scrapy 开发的，它将前面提及的 Scrapy 技术中大量需要人工重复的工作可视化、简单化。所有工具都不是万能的，在简化开发的同时对某些高级功能（例如，反爬技术）单独使用 Portia 就有所欠缺。Scrapinghub 也为此专门开发了其他的产品以构成完整的爬网生态。当然，在成本许可的范围内，也可以尝试付费使用他们的爬网方案，毕竟 Scrapinghub 可谓是爬网领域的先驱了。

## 安装Portia

Portia 提供了多种安装方法，推荐使用 Docker 来安装，这个方法是各种方案中最简单快捷的。首先确保机器上已安装并运行了 Docker。然后通过 Docker 拉一个 scrapinghub/portia 的镜像到本地，可以运行以下指令来查找 Portia 的镜像：

```
$ docker search portia
```

上面指令的输出结果如下：

```
$ docker search portia
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
scrapinghub/portia			11	
scrapinghub/portia-on-dash			5	[OK]
scrapinghub/scrapinghub-stack-portia			4	
vimagick/portia	Visual scraping for Scrapy	4		[OK]
sayden/portia	Portia is an awesome open source new web-s...	3		
sayden/portia-trusted		3		[OK]
brucetang/portia	this is the useful tool for crawl the page...	1		
praekeltfoundation/portia	Phone number lookups	0		[OK]
helloaipacino/portia	portia	0		[OK]



scrapinghub/kumo-stack-portia		0	
portiad/sqlitebrowser	DB Browser for SQLite in a container	0	[OK]
smtx/portia		0	[OK]
starjason/portia	from https://github.com/scrapinghub/portia	0	[OK]
sntran/portia	Visual scraping for Scrapy	0	[OK]
englneer/portia		0	
brianfusionex/portia	Portia is a tool that allows you to visual...	0	
portiad/hello-docker	docker in action test	0	[OK]
siegfried415/portia-dashboard	auto build portia-dashboard docker image f...	0	[OK]
2012summerain/portia		0	
devno0b/portia		0	
zchome/portia		0	
portiad/hello-dockerfile		0	
zhangqijun18/portia		0	
chenyw123/portia		0	
dataworks/portia		0	

第一项 scrapinghub/portia 是 Scrapinghub 发布的镜像，直接使用这个，指令如下：

```
$ docker pull scrapinghub/portia
```

然后运行 Portia 的容器实例，指令如下：

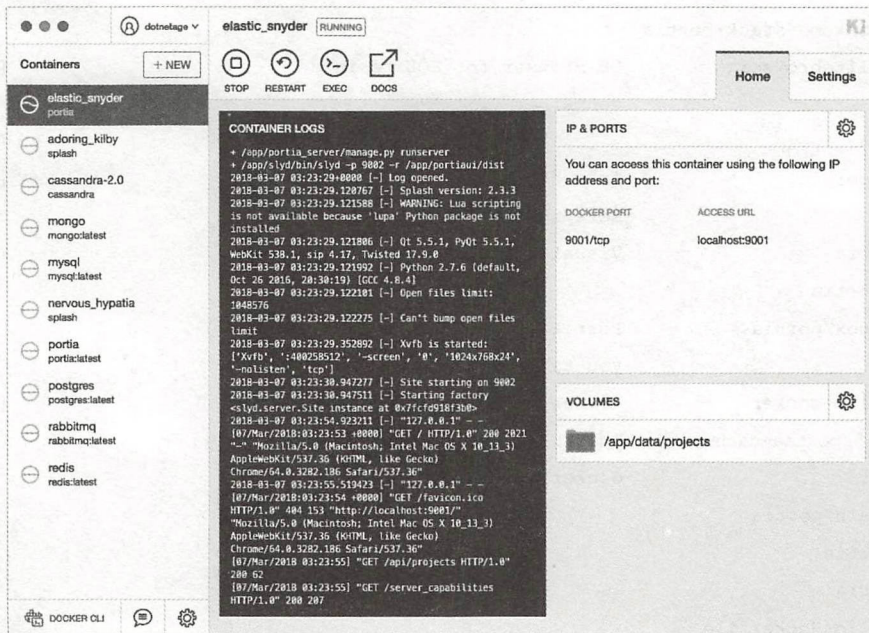
```
$ docker run -v ~/portia_projects:/app/data/projects:rw -p 9001:9001
scrapinghub/portia
```

以上指令的第一参数-v 是绑定虚拟机中的项目目录与本机目录，第二个参数-p 是虚拟机与本机端口的映射，最后的参数是原有镜像的名称。

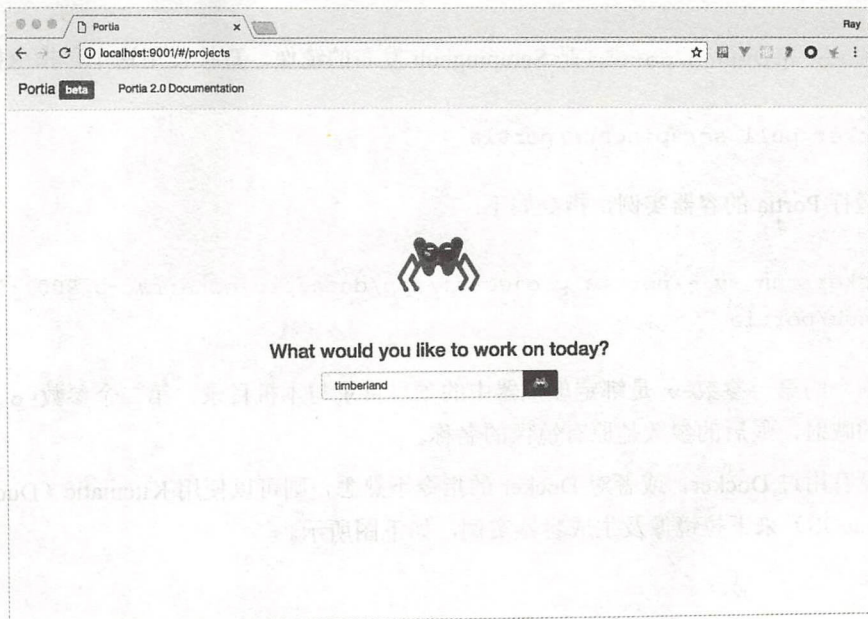
如果没有用过 Docker，或者对 Docker 的指令不熟悉，则可以使用 Kitematic（Docker 的 UI 工具 for macOS）来下拉镜像及生成容器实例，如下图所示。







当成功运行 Portia 容器后，在浏览器输入 `http://localhost:9001` 就会直接进入 Portia 的运行界面，如下图所示。



### 5.4.1 示例：某点评网爬虫

接下来会用 Portia 构建一个点评网爬虫作为示例，以实践的方式来了解 Portia 的用法。这个示例是从某点评网上收集广州中心城区珠江新城附近人气最高的美食店铺。

除了以上命令，我们还需要将高级虫术融入其中，具体如下：

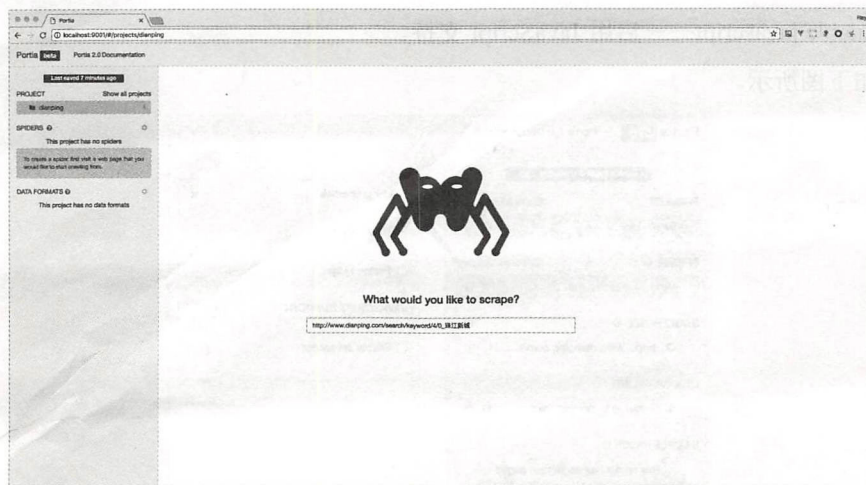
- 支持 URL 去重；
- 支持随机 UA；
- 支持随机代理。

打开浏览器 `http://localhost:9001`，在 Portia 的欢迎页面新建一个名为 `dianping` 的项目。然后 Portia 会自动转入下一个页面并要求你输入一个地址，这个地址是 Portia 首次打开的一个默认页面，打开此 URL 后，Portia 并不会进行任何其他操作，它只是需要一个入口。

我们要找到广州珠江新城附近的店铺，只需要使用的搜索页面就可以收集到数据了，搜索地址如下：

```
http://www.网址/search/keyword/4/0_珠江新城
```

在 Portia 的地址页栏输入以上 URL，如下图所示。



输入 URL 后，Portia 会在右边的浏览窗口中显示该页的内容。这个浏览窗口中的 URL 是可以改动的，当前的 URL 并不属于任何蜘蛛，只是用作普通浏览之用。接下来就可以创建蜘蛛了，单击 URL 栏的右边 `New Spider` 的按钮就将当前页面作为新建蜘蛛的搜索域（`allow_domains`），如下图所示。



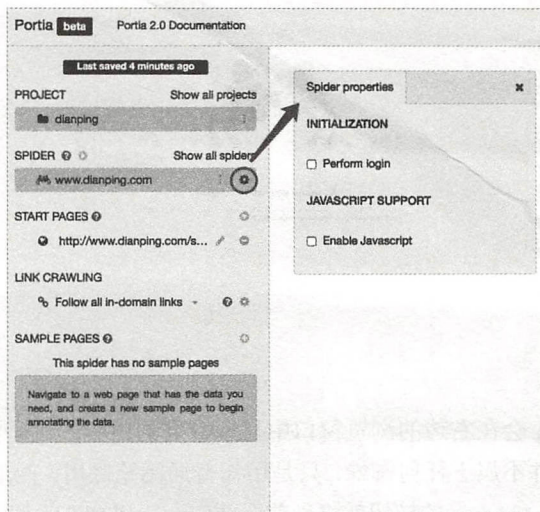




创建蜘蛛后，会在左边的导航面板的 Spider 栏位下出现 `www.网址.com` 项，这是 Portia 以当前页面的域名来命名的新蜘蛛。单击该项右边的齿轮图标，可以展开蜘蛛的项目基本特性：

- Preform login——在爬取前先进行用户登录操作；
- Enable javascript——启用 JavaScript 支持。

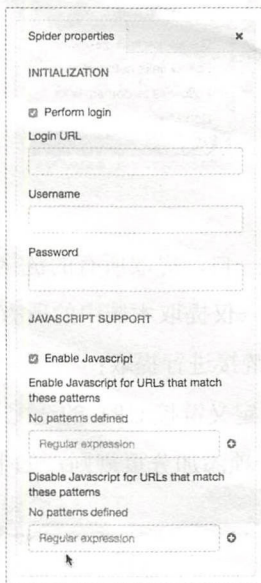
具体如下图所示。



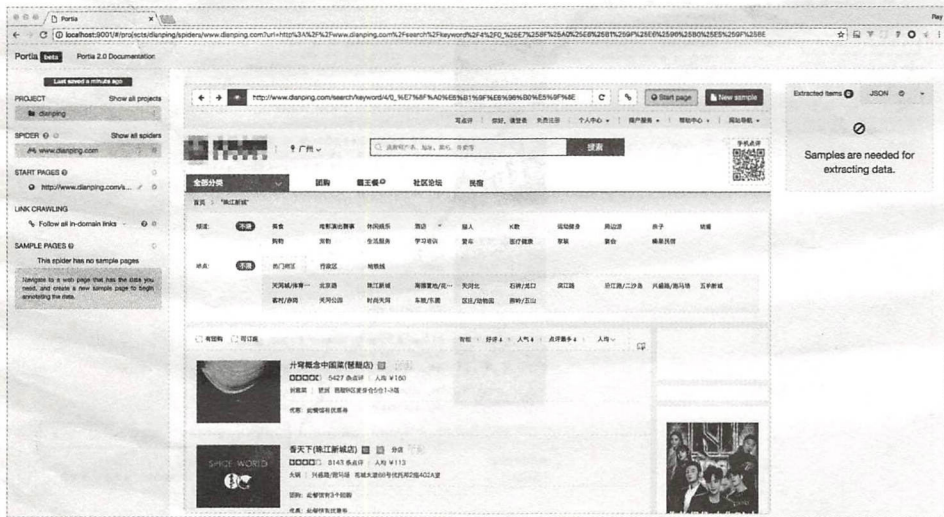
当选定并启用以上两个选项后，需要输入登录地址、用户名、密码，以及启用 JavaScript



的 URL 的适配规则，如下图所示。



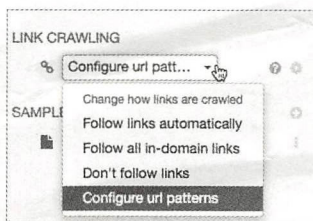
接下来为蜘蛛指定起始爬取地址(start\_urls)。单击浏览窗口的地址栏右侧的 Start Page，就会将当前页面作为起始地址自动添加到蜘蛛中，如下图所示。



这是一个常见的搜索结果分页形式的页面，我们可以采用前面曾讲述过的 Scrapy 的链接提取器来跟进爬取下一个页面以实现自动分页提取。此时就可以单击 Portia 左侧操作面板的 Link crawling 链接，这是一个多选项，下拉后会出现如下图所示的几个选项。



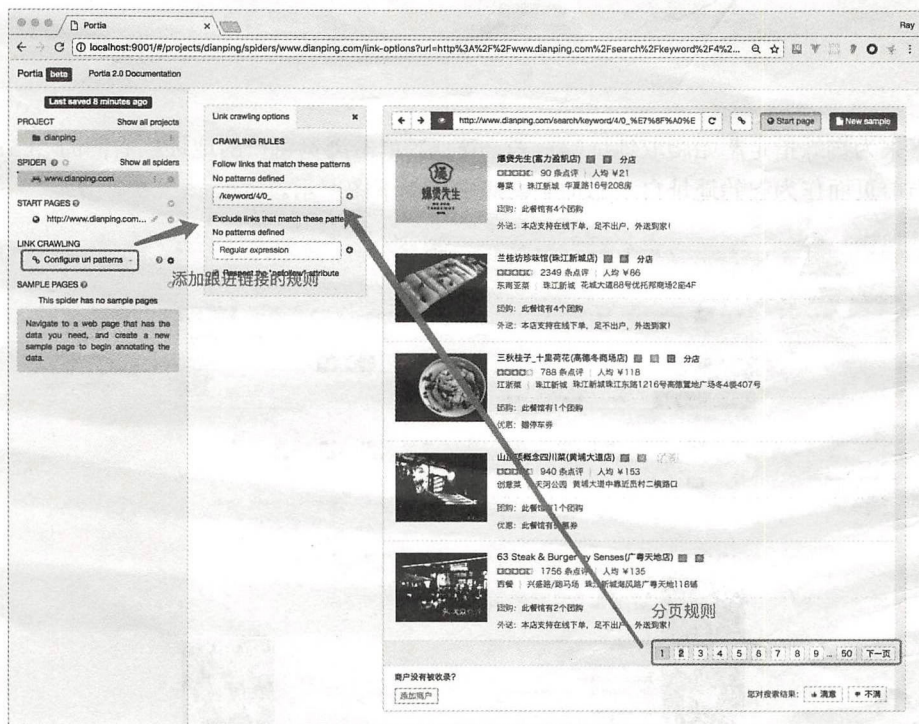




具体解释如下:

- Follow links automatically——自动提取所有的链接（任意域）;
- Follow all in-domain links——仅提取本域中的所有链接;
- Don't follow links——不对链接进行提取;
- Configure url patterns——自定义链接 URL 的适配规则。

根据实际需要, 此处选择最后一项添加分页规则, 如下图所示。



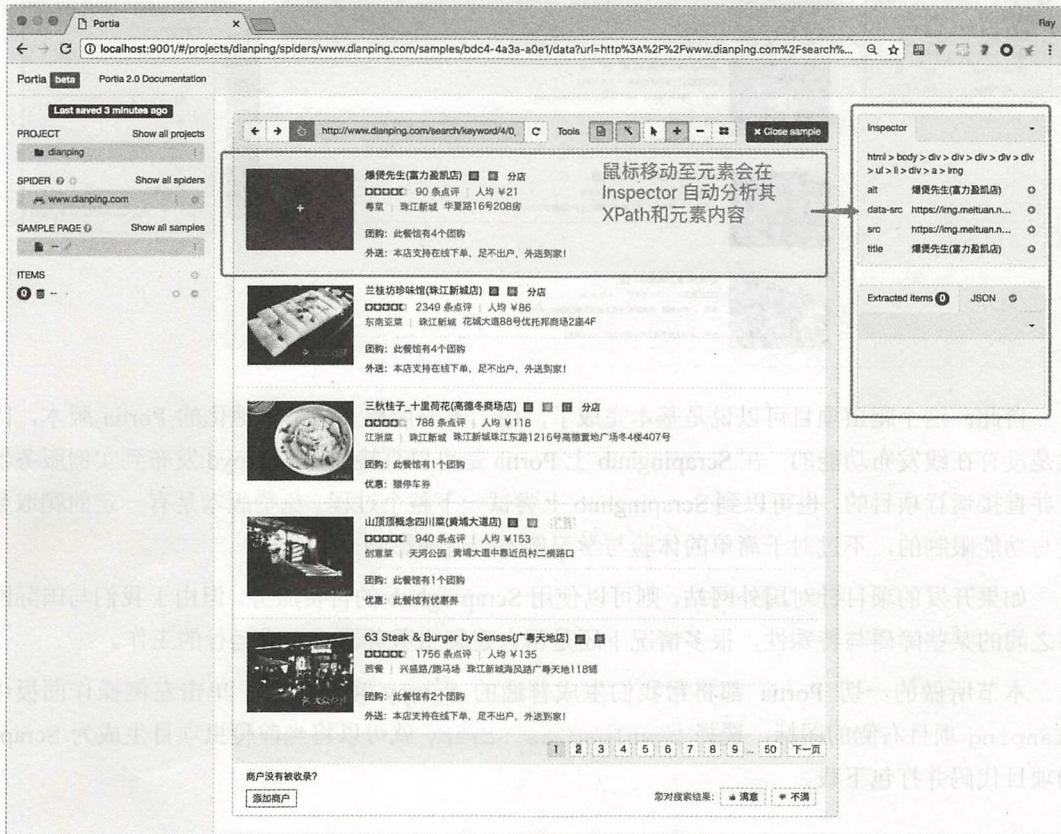
规则很简单, 就是链接中包含 `/keyword/4/0` 字符串的就进行提取。这个分析办法前面曾介绍过, 只要仔细观察分页栏的链接内容就可以得出, 此处不再赘述。

一个爬虫项目最耗费时间与精力的无疑是设计数据项目 (Item), 以及写出从页面提取数据



项的 XPath 或者 CSS 选择器，这得“一个萝卜一个坑地刨”。Portia 最让人用得“舒坦”的地方就是通过简单直观的用户界面为我们简化了这一痛苦的过程！

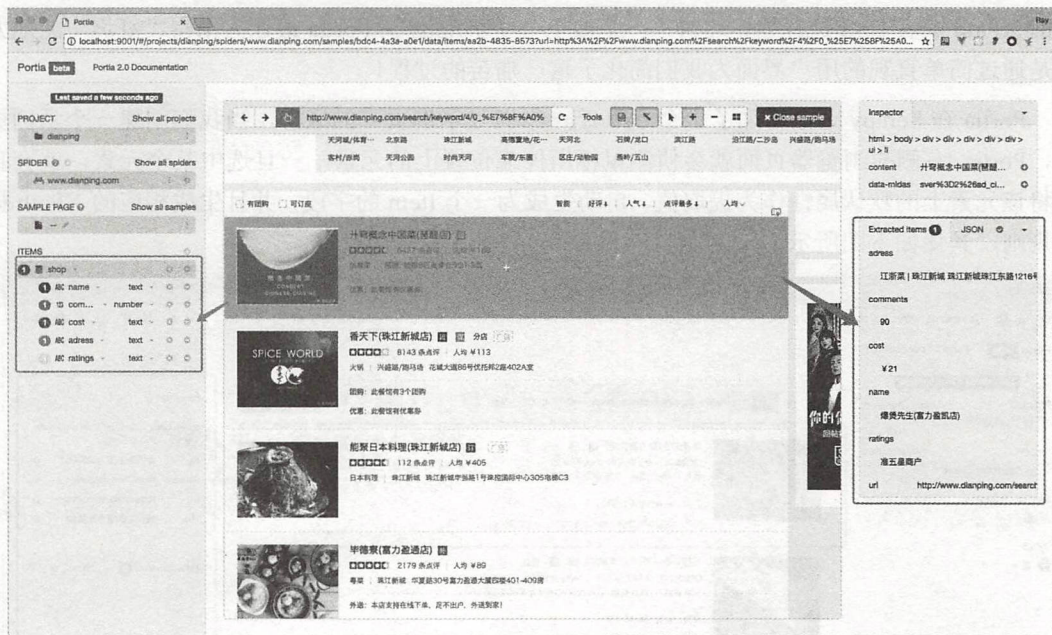
Portia 在 Scrapy 之上增加了一个 Sample Page（示例页）的概念，当我们创建一个示例页时，Portia 右侧的浏览器页面就会侦测鼠标所停靠位置上的元素，一旦选中某个元素，就会直接将该元素上的默认属性作为提取值，自动生成为一个 Item 的字段，并且生成相应的 XPath 提取规则，具体如下图所示。



Portia 对 Scrapy 的字段类型进行了扩展，使其可以支持多种常规的数据类型，而不再是原来的动态类型了。当建立了 Item 与页面元素的映射关系后，只要鼠标在某个元素上停留，Portia 都会自动地按照 Item 与元素的映射关系进行提取预览，如下图所示。







自此, 这个爬虫项目可以说是基本完成了, 由于本书所提及的是本地化的 Portia 版本, 因此是没有在线发布功能的。在 Scrapinghub 上 Portia 是可以直接通过 Scrapy 发布到实例服务器上并直接运行项目的, 也可以到 Scrapinghub 上尝试一下整个过程。免费版本是有一定的爬取数量与功能限制的, 不过对于简单的体验与学习倒不会造成影响。

如果开发的项目针对国外网站, 则可以使用 Scrapinghub 的付费服务, 但由于我们与国际网络之间的某些障碍与特殊性, 很多情况下还是得自建爬虫服务器来完成运行的工作。

本节所做的一切 Portia 都将帮我们生成普通的 Scrapy 项目代码, 单击左侧操作面板中 dianping 项目右侧的图标, 选择 Download as Scrapy 就可以将当前爬虫项目生成为 Scrapy 的项目代码并打包下载。

## 5.4.2 解读Portia爬虫代码

解压 Portia 生成的项目压缩包, 其项目结构如下:

```
.
├── Dianping
│   ├── __init__.py
│   └── items.py
```

```

|   ├── middleware.py
|   ├── pipelines.py
|   ├── settings.py
|   ├── spiders
|   |   ├── __init__.py
|   |   └── dianping_com.py
|   └── utils
|       ├── __init__.py
|       ├── parser.py
|       ├── processors.py
|       ├── spiders.py
|       └── starturls.py
└── scrapy.cfg
└── setup.py

```

Portia 生成的项目结构与本书中介绍的内容基本没有太大的差别，只是多出了一个叫 `utils` 的包，这个包会留到最后来解读它。

`middleware.py`、`pipelines.py` 和 `settings.py` 都是由 `scrapy startproject` 指令构建的，其内容没有任何改变。下面就从 `items.py` 入手，来看看 Scrapy 官方是怎么写 Item 的，从中我们也学习一些相关的技巧。

`items.py` 的具体代码如下所示。

```

from __future__ import absolute_import

import scrapy
from collections import defaultdict
from scrapy.loader.processors import Join, MapCompose, Identity
from w3lib.html import remove_tags
from .utils.processors import Text, Number, Price, Date, Url, Image

class PortiaItem(scrapy.Item):
    fields = defaultdict(
        lambda: scrapy.Field(
            input_processor=Identity(),
            output_processor=Identity()
        )
    )

```



```
def __setitem__(self, key, value):
    self._values[key] = value

def __repr__(self):
    data = str(self)
    if not data:
        return '%s' % self.__class__.__name__
    return '%s(%s)' % (self.__class__.__name__, data)

def __str__(self):
    if not self._values:
        return ''
    string = super(PortiaItem, self).__repr__()
    return string
```

```
class ShopItem(PortiaItem):
    name = scrapy.Field(
        input_processor=Text(),
        output_processor=Join(),
    )
    cost = scrapy.Field(
        input_processor=Text(),
        output_processor=Join(),
    )
    ratings = scrapy.Field(
        input_processor=Text(),
        output_processor=Join(),
    )
    adress = scrapy.Field(
        input_processor=Text(),
        output_processor=Join(),
    )
    comments = scrapy.Field(
        input_processor=Number(),
        output_processor=Join(),
    )
```

这里的代码只有两个类，一个是 `PortiaItem` 类，另一个是继承自该类的 `ShopItem`，其中包含了在 `Portia` 的 Web 界面中定义的字段与类型。

`PortiaItem` 是对 `scrapy.Item` 的一个简单扩展，一是增加了 `fields` 的枚举字段，将所有自定义字段包含其中；二是进行了输出增强，使得调用 `to_str` 方法时可以看到更多的内容。

`ShopItem` 的定义是在 `PortiaItem` 的基础上对每个字段的类型增加了序列化操作，请留意 `Portia` 构造 `Field` 的方式：

```
scrapy.Field(
    input_processor=Text(),
    output_processor=Join(),
)
```

通过指定不同的输入处理器函数与输出处理器函数实现了强类型的转换。而类型的定义都源自于 `utils.processors` 包。

打开这个包，会看到里面有很多基本处理器的定义，而全部的处理器都继承自一个 `BaseProcessor` 类，这个类会对字段值进行一些修正的预处理。以 `Text()` 输入处理器为例：

```
class Text(BaseProcessor):
    def __call__(self, values):
        return [remove_tags(v).strip()
                if v and isinstance(v, six.string_types) else v
                for v in values]
```

这个类就是将文本中多余的网页标记去除，以得到“纯正”的文本内容。其他字段的定义也大同小异，都是将文本内容先进行格式化，然后进行类型转换。由于这些代码都可以通过 `Portia` 生成，并不需要进行编码处理，此处就不一一解释了，有兴趣读者可以细读其中的代码。

打开 `spiders.dianping_com.py`，看一下 `Portia` 生成的蜘蛛：

```
from __future__ import absolute_import

from scrapy import Request
from scrapy.linkextractors import LinkExtractor
from scrapy.loader import ItemLoader
from scrapy.loader.processors import Identity
from scrapy.spiders import Rule
```



```

from ..utils.spiders import BasePortiaSpider
from ..utils.starturls import FeedGenerator, FragmentGenerator
from ..utils.processors import Item, Field, Text, Number, Price, Date, Url,
Image, Regex
from ..items import PortiaItem

class Dianping(BasePortiaSpider):
    name = "www.dianping.com"
    allowed_domains = [u'www.dianping.com']
    start_urls = [
        u'http://www.dianping.com/search/keyword/4/
0_%E7%8F%A0%E6%B1%9F%E6%96%B0%E5%9F%8E']
    rules = [
        Rule(
            LinkExtractor(
                allow=(u'/keyword/4/0_'),
                deny=()
            ),
            callback='parse_item',
            follow=True
        )
    ]
    items = [[Item(PortiaItem,
        None,
        u'#shop-all-list > ul',
        [Field(u'ratings',
            'li:nth-child(5) > .txt > .comment > .sml-rank-stars::
attr(title)',
            []),
        Field(u'name',
            'li:nth-child(13) > .pic > a > img::attr(title)',
            []),
        Field(u'comments',
            'li:nth-child(13) > .txt > .comment > .review-num >
b *::text',
            []),
        Field(u'cost',

```

```

        'li:nth-child(13) > .txt > .comment > .mean-price >
b *::text',
        [],
        Field(u'adress',
        'li:nth-child(15) > .txt > .tag-addr *::text',
        []))
    ]]

```

Portia 使用了它扩展的蜘蛛作为基类，与我们前面接触的标准蜘蛛不同的是，这里没有 parse 方法。但在爬取规则中，回调方法是被指向名为 parse\_item 的函数，这明显是调用基类 BasePortiaSpider 的分析函数进行数据提取与 Item 的赋值的。后面我们再转跳到这个方法来解读这种公用的提取方式。

这里有一个 items 的属性，它承载的是一个双重枚举类型的值。这里的 Item 并不是 Scrapy 的原生 Item 类，而是一个 utils.processors.Item 的处理器类，用于存储当前 Item 的类型、选择器与数据结构。从代码中就很容易读出这里就是字段值与 XPath 的映射关系。到底 BasePortiaSpider 是如何将这个映射关系转化为 Item 实例的呢？这个非常值得研究，学会了这招就能开发出像 Portia 一样的通用蜘蛛了。

打开 utils.spiders 一探究竟，找到 BasePortiaSpider，其代码如下所示。

```

class BasePortiaSpider(CrawlSpider):
    items = []

    def start_requests(self):
        for url in self.start_urls:
            if isinstance(url, dict):
                type_ = url['type']
                if type_ == 'generated':
                    for generated_url in FragmentGenerator()(url):
                        yield self.make_requests_from_url(generated_url)
                elif type_ == 'feed':
                    yield FeedGenerator(self.parse)(url)
            else:
                yield self.make_requests_from_url(url)

    def parse_item(self, response):
        for sample in self.items:

```



```
        items = []
        try:
            for definition in sample:
                items.extend(
                    [i for i in self.load_item(definition, response)]
                )
        except RequiredFieldMissing as exc:
            self.logger.warning(str(exc))
        if items:
            for item in items:
                yield item
            break

    def load_item(self, definition, response):
        query = response.xpath if definition.type == 'xpath' else response.css
        selectors = query(definition.selector)
        for selector in selectors:
            selector = selector if selector else None
            ld = PortiaItemLoader(
                item=definition.item(),
                selector=selector,
                response=response,
                baseurl=get_base_url(response)
            )
            for field in definition.fields:
                if hasattr(field, 'fields'):
                    if field.name is not None:
                        ld.add_value(field.name,
                                    self.load_item(field, selector))
                elif field.type == 'xpath':
                    ld.add_xpath(field.name, field.selector, *field.processors,
                                required=field.required)
                else:
                    ld.add_css(field.name, field.selector, *field.processors,
                               required=field.required)
            yield ld.load_item()
```

BasePortiaSpider 继承于 CrawlSpider, 增加了 items 的数组类型的字段, 增加了

parse\_item 和 load\_item 函数，并且重写了 start\_requests 函数。

start\_requests 函数的功能不变，只是调整了生成起始爬取 URL 的规则，重点在于 parse\_item 和 load\_items 两个函数。

首先分析函数 parse\_item:

```
def parse_item(self, response):
    for sample in self.items:
        items = []
        try:
            for definition in sample:
                items.extend(
                    [i for i in self.load_item(definition, response)]
                )
        except RequiredFieldMissing as exc:
            self.logger.warning(str(exc))
        if items:
            for item in items:
                yield item
            break
```

这个函数将 items 中的字段映射关系作为样本提取出单个的 Item 定义，这样就能让一个蜘蛛同时支持多种不同数据结构的 Item。由于可能出现不同的数据结构，那么页面元素可能由于缺失而产生 Null 值，从而引发异常。因此通过 try 结构来保证每个循环能正常地被执行，即使发生异常也只是出现警告性的日志记录。最后返回 item 的枚举。

这里就引发了另一个关键性的函数调用：

```
self.load_item(definition, response)
```

看一下 load\_item 函数的具体定义：

```
def load_item(self, definition, response):
    query = response.xpath if definition.type == 'xpath' else response.css
    selectors = query(definition.selector)
    for selector in selectors:
        selector = selector if selector else None
        ld = PortiaItemLoader(
```



```

        item=definition.item(),
        selector=selector,
        response=response,
        baseurl=get_base_url(response)
    )
    for field in definition.fields:
        if hasattr(field, 'fields'):
            if field.name is not None:
                ld.add_value(field.name,
                             self.load_item(field, selector))
            elif field.type == 'xpath':
                ld.add_xpath(field.name, field.selector, *field.processors,
                             required=field.required)
            else:
                ld.add_css(field.name, field.selector, *field.processors,
                           required=field.required)
    yield ld.load_item()

```

这个函数的逻辑非常简单，分三步：

(1) 从 `definition (utils.processors.Item)` 中获取选择器的类型，如果没有被指定，则默认采用 CSS 选择器。

(2) 生成一个 `ItemLoader` 并将数据-元素映射关系通过 `add_value`、`add_xpath` 或 `add_css` 加入 `ItemLoader`。

(3) 调用 `ItemLoader.load_item` 方法将数据一次性加载并生成 `Item` 的枚举实列返回。

这里出现了一个 `ItemLoader` 的概念，直接查看 `Scrapy` 的官方文件也会找到它，如果直接使用它，则会让代码非常难读。`Scrapinghub` 对该类的解释也是讳莫如深，最初接触 `Scrapy` 时我也一直存在这样的困惑，到底 `ItemLoader` 的存在有何意义？

这个答案却在 `Portia` 生成的代码中找到了，`ItemLoader` 是一个泛用型关系数据-元素加载器，当我们要将这种在每个项目中不断重复的关系与值之间的映射代码统一化时，它才会显现威力。

### 5.4.3 数据项加载器——Item Loaders

`Item Loaders` 提供了一种简便的构件 (mechanism) 来抓取 `ref:Items<topics-items>`。虽然 `Items` 可以从它自己的类似字典 (dictionary-like) 的 API 中得到所需信息，但 `Item Loaders`

提供了许多更加方便的 API，这些 API 自动完成那些具有共通性的任务，从抓取进程中得到这些信息，比如预先解析提取到的原生数据。换句话说，Items 提供了盛装抓取到的数据的容器，而 Item Loaders 则提供了一种将数据装入容器的执行动作。

Item Loaders 被设计用来提供一个既弹性又高效简便的构件，以扩展/重写爬虫或源格式（HTML、XML 之类的）等区域的解析规则。

### 用 Item Loaders 装载 Items

要使用 Item Loader，必须先将它实例化。可以使用类似字典的对象（例如，Item、dict）来进行实例化，或者不使用对象也可以，当不用对象进行实例化时，Item 会自动使用 ItemLoader.default\_item\_class 属性中指定的 Item 类在 Item Loader constructor 中实例化。

在开始收集数值到 Item Loader 中时，通常使用 Selectors。可以在同一个 item field 中添加多个数值，Item Loader 知道如何用合适的处理函数来“添加”这些数值。

下面是 Spider 中典型的 Item Loader 的用法，使用 Items chapter 中声明的 Product item：

```
from scrapy.loader import ItemLoader
from myproject.items import Product

def parse(self, response):
    l = ItemLoader(item=Product(), response=response)
    l.add_xpath('name', '//div[@class="product_name"]')
    l.add_xpath('name', '//div[@class="product_title"]')
    l.add_xpath('price', '//p[@id="price"]')
    l.add_css('stock', 'p#stock')
    l.add_value('last_updated', 'today') # you can also use literal values
    return l.load_item()
```

快速查看这些代码之后，可以看到 name 字段从页面中两个不同的 XPath 位置被提取出来了：

```
//div[@class="product_name"]
//div[@class="product_title"]
```

换句话说，数据通过 add\_xpath() 方法把从两个不同的 XPath 位置提取的数据收集起来了。这是以后分配给 name 字段中的数据。

之后，类似的请求被用于 price 和 stock 字段（后者使用 CSS selector 和 add\_css() 方法），最后使用不同的 add\_value() 方法对 last\_update 填充文本值（today）。



最终, 当所有数据被收集起来之后调用 `ItemLoader.load_item()` 方法, 此时 `Item Loader` 才真正地将数据填充到 `Item` 实例 (`Product`) 中, 并将其返回。

### 输入与输出处理器

`Item Loader` 在每个 (`Item`) 字段中都包含了一个输入处理器和一个输出处理器。输入处理器收到数据时立刻提取数据 (通过 `add_xpath()`、`add_css()` 或者 `add_value()` 方法), 之后输入处理器的结果被收集起来并保存在 `ItemLoader` 中。收集到所有数据, 调用 `ItemLoader.load_item()` 方法来进行填充, 并得到填充后的 `Item` 对象。

其实这就是上一节 `load_item` 函数中 `PortiaItemLoader` 所进行的处理:

```
for field in definition.fields:
    if hasattr(field, 'fields'):
        if field.name is not None:
            ld.add_value(field.name,
                          self.load_item(field, selector))
        elif field.type == 'xpath':
            ld.add_xpath(field.name, field.selector, *field.processors,
                          required=field.required)
        else:
            ld.add_css(field.name, field.selector, *field.processors,
                       required=field.required)
```

用另一种简化的形式来说明 (以下为伪代码):

```
l = ItemLoader(Product(), some_selector)
l.add_xpath('name', xpath1) # (1)
l.add_xpath('name', xpath2) # (2)
l.add_css('name', css) # (3)
l.add_value('name', 'test') # (4)
return l.load_item() # (5)
```

上述伪代码做了这些事情:

(1) 从 `xpath1` 提取出的数据传递给输入处理器的 `name` 字段。输入处理器的结果被收集和保存在 `Item Loader` 中 (但尚未分配给该 `Item`)。

(2) 从 `xpath2` 提取出来的数据传递给 `l` 中使用的相同的输入处理器。输入处理器的结果被附加到 `l` 中收集的数据 (如果有的话) 中。

(3) 这个例子与上一例相似, 除了它使用的是 `CSS` 选择器。

(4) 这个例子是直接将值绑定到指定的字段 `name` 中，而不是通过执行选择器。在 `load_item()` 中赋值。

(5) 将 1~4 的数据收集起来，将每个字段内容传入输出加载器，最后将其值一次性加载到 `Item` 实例中。

输入/输出处理其实就相当于两个不同方向的函数管道，我们可以对输入与输出进行不同方式的格式化，或者转换处理。最终由 `load_item` 一次性执行它们而得到 `Item` 实例。

## 5.4.4 最后的工作

由于 `Portia` 生成的 `settings.py` 文件并没有加入其他配置项目，我们只需要将“代理池”一节中的 `RandomProxyMiddleware` 加入配置中，使其支持随机代理即可。同样，将“客户端仿真”一节中介绍的 `RandomUserAgentMiddleware` 类加入配置中以支持随机 UA，最后添加 `scrapy.dupefilters.RFPDupeFilter` 进行 URL 去重，这个配置一定要加上，否则分页爬取会进入死循环而失败。

具体配置如下：

```
BOT_NAME = 'Dianping'

SPIDER_MODULES = ['Dianping.spiders']
NEWSPIDER_MODULE = 'Dianping.spiders'
ROBOTSTXT_OBEY = True

# 重写默认的请求头
DEFAULT_REQUEST_HEADERS = {
    'User-Agent': 'Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.36 (KHTML,
like Gecko) Ubuntu Chromium/60.0.3112.113 Chrome/60.0.3112.113 Safari/537.36',
    'Accept':
'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
}

# Enable or disable downloader middlewares
# See http://scrapy.readthedocs.org/en/latest/topics/downloader-middleware.html
DOWNLOADER_MIDDLEWARES = {
    'Dianping.middlewares.RandomUserAgent': 543,
    'Dianping.middlewares.RandomProxyMiddleware': 801
}
```



```

}
DUPEFILTER_CLASS = "scrapy.dupefilters.RFPDupeFilter"

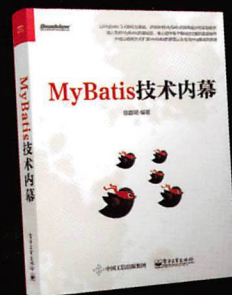
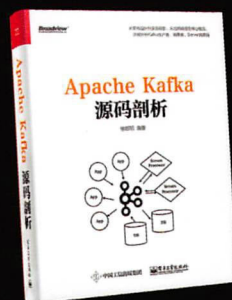
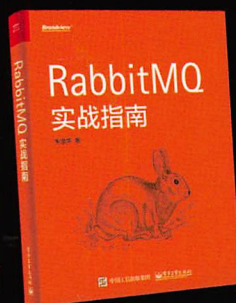
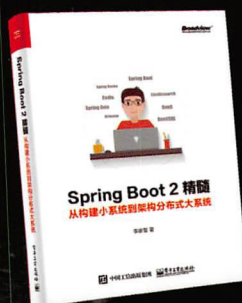
# 设置存储后端
FEED_FORMAT='json'
FEED_URI = 'results.json'

USER_AGENTS = [
    "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET CLR 2.0.50727; Media Center PC 6.0)",
    "Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; WOW64; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET CLR 1.0.3705; .NET CLR 1.1.4322)",
    "Mozilla/4.0 (compatible; MSIE 7.0b; Windows NT 5.2; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.2; .NET CLR 3.0.04506.30)",
    "Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN) AppleWebKit/523.15 (KHTML, like Gecko, Safari/419.3) Arora/0.3 (Change: 287 c9dfb30)",
    "Mozilla/5.0 (X11; U; Linux; en-US) AppleWebKit/527+ (KHTML, like Gecko, Safari/419.3) Arora/0.6",
    "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.2pre) Gecko/20070215 K-Ninja/2.1.1",
    "Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9) Gecko/20080705 Firefox/3.0 Kapiko/3.0",
    "Mozilla/5.0 (X11; Linux i686; U;) Gecko/20070322 Kazehakase/0.4.5",
    'Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.36 (KHTML, like Gecko) Ubuntu Chromium/60.0.3112.113 Chrome/60.0.3112.113 Safari/537.36',
    'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36',
    'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.104 Safari/537.36 Core/1.53.2372.400 QQBrowser/9.5.11096.400',
    'Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/20100101 Firefox/55.0'
]

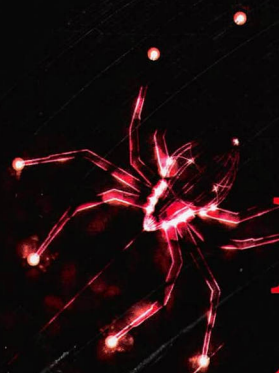
```

完成以上工作就能将 Portia 生成的点评爬虫项目加载运行了。

## 好书分享







# Python绝技

## ◆爬虫初步

提供学习虫术的技术线路图，介绍爬虫基本的实现方法与实际运用。

## ◆Scrapy基础

以Scrapy架构为核心，详解Scrapy架构和各个模块的作用。

## ◆Scrapy工程管理与部署

详解Scrapyd的安装配置，介绍scrapy-client和scrapy-deploy的使用方法。

## ◆中阶虫术

分析Scrapy的蜘蛛内部实现，运用Selenium和Splash处理棘手的JavaScript网页，详解如何处理采集后的数据。

## ◆高阶虫术

聚焦于爬虫系统的性能，讲解如何能让爬虫变得更加隐蔽，如何让爬虫能看懂图片并训练它们使之变得更加聪明。讲解虫术的“大招”（分布式爬虫）来应对大规模的数据采集工作与数据存储工作。



博文视点Broadview



@博文视点Broadview



责任编辑：陈晓猛  
封面设计：李玲

上架建议：计算机 / 大数据

ISBN 978-7-121-34456-5



9 787121 344565 >

定价：99.00元